UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Administrdor de presentación para jPOS (plataforma Java para transacciones electrónicas.)
Autores: Jerónimo Paoletti
Director: Claudia Queiruga
Codirector: Laura Fava
Carrera: Licenciatura en Sistemas

Resumen

En los últimos años, una de las áreas de negocios que más se ha expandido es la de las transacciones electrónicas financieras. En este marco ha aparecido jPOS, una biblioteca / framework OpenSource desarrollado en Java y basado en el protocolo ISO-8583, que puede ser personalizado y extendido para implementar transacciones electrónicas de todo tipo (incluyendo financieras). Los proyectos de esta índole suelen estar acompañados de información complementaria vinculada al negocio abordado. Dicha información debe ser administrada y gestionada desde una interfaz de usuario y su desarrollo puede ser tan costoso de realizar como el sistema transaccional propiamente dicho (o incluso más). Esto hace que se multiplique el trabajo de los programadores y que la atención sea desviada del objetivo principal. Es por esto que una herramienta integrada a jPOS, de fácil uso, integración y expansión puede ser de utilidad para reducir el desvío de la atención y la carga de desarrollo del proyecto, disminuyendo así costos y plazos.

Palabras Claves

JPOS, Administración de Presentación, OpenSource, Transacciones electrónicas, Monitoreo transaccional, Java.

Conclusiones

Este proyecto ha resultado altamente enriquecedor a nivel personal y profesional tanto desde el punto de vista de las tecnologías utilizadas como desde el punto de vista de la integración al mundo OpenSource. El trabajo se ha aplicado en un proyecto real exitosamente y la metodología ha sido adoptada por varios desarrolladores, quienes resultaron satisfechos.

Trabajos Realizados

Los resultados tangibles de este trabajo fueron seis módulos para jPOS, una API de desarrollo para los programadores usuarios de jPOS, una guía rápida en inglés y un sitio de alojamiento público (GitHub) para que el proyecto sea continuado por una comunidad OpenSource.

Trabajos Futuros

Se espera continuar con mejoras y agregados sobre el producto resultante, obtener apoyo de la comunidad OpenSource de jPOS e implementar nuevos módulos que ofrezcan a los desarrolladores más herramientas y facilidades para extender los beneficios obtenidos por el uso de este trabajo.

Fecha de la presentación: Octubre 2010

Tesina de Licenciatura en Sistemas

jPOS Presentation Manager

Jerónimo Paoletti

Octubre 2010

Índice de contenido

Índice del figuras

Índice de tablas

Capitulo 1. Introducción

1.1. Contexto y motivación general

En un mundo donde las tecnologías de la comunicación evolucionan rápidamente, los negocios sustentados por la comunicación electrónica aparecen a diario. Un conjunto importante de estos negocios están relacionados con entidades financieras y con dispositivos terminales utilizados para llegar al usuario final y que, a través de las **transacciones electrónicas**, les ofrecen diversos servicios. Con el fin de unificar la amplia gama de protocolos surgidos de esta necesidad de comunicación se desarrolló el protocolo estándar de transacciones electrónicas financieras ISO-8583[1] (ver también [2]).

En este contexto surge jPOS[17], un producto OpenSource hecho en Java, pensado para realizar todo tipo de transacciones electrónicas basadas en dicho protocolo. Con un diseño modular flexible y extensible, jPOS permite una rápida adaptación a una amplia gama de negocios basados en transacciones electrónicas. Gracias a esto y a su filosofía OpenSource, se formó una comunidad que respalda el producto y que ha creado varios módulos optativos fácilmente integrables.

Un breve trabajo de campo de investigación de sistemas existentes, el encuestado de expertos en el tema y la experiencia personal del autor de este proyecto dejaron en evidencia que, si bien el foco de dichos sistemas (basados en jPOS) son las transacciones, la mayor parte de los sistemas tienen un modelo de negocios complejo asociado, con numerosas entidades administrativas que deben ser gestionadas y que le dan sentido y contexto a la transacción.

La investigación también reveló que los desarrolladores usuarios de jPOS suelen tener problemas para desarrollar las interfaces administrativas -o directamente evitan hacerlas- porque su especialización es el desarrollo de módulos transaccionales y realizar dichas interfaces suele demandar conocimientos profundos en áreas a las que no están habituados.

Presentado el problema, se desprende la necesidad de un módulo integrado a jPOS que permita generar una interfaz administrativa rápidamente y de forma simple pero extensible, de forma tal que se pueda empezar con algo básico que acompañe al desarrollo del núcleo transaccional pero que permita luego poder mejorarla sin cambios radicales.

1.2. Motivación personal: una necesidad laboral concreta

Además de la motivación general presentada anteriormente, hay también una necesidad personal del autor de este trabajo dado el surgimiento de un proyecto concreto en el ámbito laborar. Dicho proyecto cuenta con todas las características adecuadas para utilizar un sistema del estilo planteado y, por ello, parte del desarrollo de este trabajo se realiza en el ámbito laboral para satisfacer también la necesidad del proyecto, lo cual le da un valor agregado importante ya que todo el desarrollo es monitorizado y validado por un cliente final real el cual presenta necesidades concretas y reales.

1.3. Investigación del contexto

El objetivo de la investigación ha sido identificar casos concretos de proyectos y negocios reales y verificar la necesidad de una herramienta que simplifique y acelere el desarrollo de interfaces de usuario complejas y completas, al mismo tiempo que ayude a los programadores a focalizar su esfuerzo en el modelo transaccional.

Para ejemplificar, algunos negocios típicos que involucran transacciones electrónicas son

- Pago de impuestos y servicios domésticos (del tipo Rapipago).
- Compras con tarjetas de débito.
- Compras con tarjetas de crédito incluyendo eventualmente financiación (cuotas).
- Sistemas de fidelización de clientes (puntos, millas, ofertas especiales, etc).
- Extracción o transferencia de dinero.
- Recarga virtual de celulares.

Prácticamente todos estos negocios tienen un modelo de negocio vinculados que, muchas veces, es invisible al usuario final. Por ejemplo, el pago con una tarjeta de crédito involucra al dueño de la tarjeta (cliente), al comercio que ofrece el producto o servicio, el proveedor de la tarjeta, la entidad financiera (banco), el producto o servicio, transferencias interbancarias, etc.

MasterCard Worldwide muestra en su sitio web oficial una interesante animación que describe claramente un ejemplo de transacción electrónica, el pago de productos o servicios a través de una tarjeta de crédito. Ver http://www.mastercard.com/za/merchant/en/how_works/index.html .

Cada una de estas entidades vinculadas al negocio deben ser modeladas en el desarrollo del software transaccional. Esto es lo que se denomina **modelo de negocios**.

1.3.1. Consultas a expertos

Como parte de la investigación, han sido consultados (vía correo electrónico) algunos referentes en materia de negocios orientados a las transacciones electrónicas (ver Anexo 1) acerca de los proyectos en los que han utilizado jPOS como herramienta principal en el desarrollo. El cuestionario realizado (en inglés) se detalla a continuación. En negrita la pregunta original y a continuación la traducción en cursiva.

1. **On how many projects have you worked using mainly jPOS?** *¿En cuantos proyectos ha usted trabajado usando principalmente jPOS?*

2. **Can you briefly mention the "business" touched by some of this projects?** *¿Puede mencionar brevemente los negocios abarcados por algunos de estos proyectos?*

3. **How many of these projects had a complex business model asociated?** *¿Cuantos de estos proyectos tuvieron un modelo de negocios complejo asociado?*

4. **How many of these projects needed extra development to build an administrative UI to handle the busisness model?** *¿Cuantos de estos proyectos necesitaron desarrollo extra para construir una interfaz administrativa para gestionar el modelo de negocio?*

5. **Was this extra development a heavy or significative job during the project?** *¿Tuvo este desarrollo extra un peso o esfuerzo significativo durante el proyeccto?*

6. **Do you think that a jPOS module to dynamic build interface (using xml and having jPOS philosophy) could be usefull to make this extra development easier or faster?** *¿Piensa usted que un módulo integrado a jPOS de construcción dinámica de interfaces (usando xml y respetando la filosofía jPOS) podría ser útil para hacer este desarrollo extra más simple o rápido?*

En líneas generales, las conclusiones son las siguientes.

- La mayoría prefiere evitar el desarrollo de la interfaz porque es costoso y complicado.

- Más de la mitad de los desarrollos poseen un modelo de negocio complejo.

- Todos estuvieron de acuerdo que una herramienta como la propuesta sería de gran utilidad en un buen número de casos.

1.3.2. Evaluación de Proyectos

Paralelamente a las consultas, han sido analizados algunos proyectos de escala provincial, nacional e internacional y sus negocios subyacentes para dejar a la vista la vinculación entre las transacciones y los modelos de negocios complejos que las acompañan y sustentan.

jCard

jCard[1] es un producto desarrollado por Alejandro Revilla -creador de jPOS- que resuelve la mayor parte de las necesidades de un sistema basado en tarjetas de crédito y débito con administración contable o financiera. El modelo de datos de jCard es una compleja interrelación entre adquirentes, comercios, terminales, cuentas, tarjetas, tarjeta-habientes, transacciones, etc. La mayoría de estas entidades deben ser administradas o monitorizadas desde una interfaz externa.

VirtualLine

VirtualLine -parte de la empresa CardLine- posee uno de los sistemas líderes en recargas virtuales de celulares en la Argentina. El sistema tiene un modelo de datos complejo que incluye distribuidores, comercios, sucursales, terminales, operadores, gateways, contratos, información contable, entre otros. A su vez, tiene una administración manual de transacciones en ejecución para recargas en situaciones particulares (que requiere interacción directa con la interfaz). El sistema tuvo que proveer una gran administración para todas estas entidades, lo cual tuvo un alto costo de desarrollo de interfaces y tiene, hoy por hoy, un alto costo de mantenimiento.

Devoto HiperCard

Devoto es la cadena de supermercados más grande de Uruguay e implementó un sistema de tarjetas para beneficios de clientes (sistema de fidelización) incluyendo descuentos, acumulación de puntos, ofertas especiales y cuotas[2]. El modelo de datos incluye clientes, tarjetas, productos, ofertas, premios, cuotas (es decir actividades financieras) entre otros, entidades que en general requieren administración.

1.4. Estructura del informe

Los capítulos subsiguientes de este informe abarcarán los siguientes temas.

Capítulo 2. jPOS: Descripción del proyecto jPOS con sus virtudes y limitaciones en el marco de proyecto OpenSource.

Capítulo 3. jPOS Presentation Manager: el desarrollo: Descripción del proceso de ingeniería aplicado en el desarrollo del producto propuesto como solución al problema.

Capítulo 4. jPOS Presentation Manager: el producto: Descripción del producto resultante desde el punto de vista de su aplicación.

Capítulo 5. Resultados y conclusiones: Descripción de los resultados obtenidos a lo largo del proyecto junto con las conclusiones.

1 http://jpos.org/blog/2010/03/what-is-jcard/

2 http://www.devoto.com.uy/mvdcms/cathomehipercard_2_1.html

Capitulo 2. jPOS

2.1. El proyecto

JPOS es un producto OpenSource, desarrollado en Java, que tiene como objetivo principal proveer un framework de desarrollo de transacciones electrónicas financieras sustentadas en el protocolo ISO8583.

Más allá de su definición formal como producto, jPOS plantea una interesante metodología de desarrollo basada en módulos integrables que luego se unen en tiempo de compilación a través de un proceso ant. Cada módulo ofrece una funcionalidad concreta pueden extenderse y utilizarse entre sí.

La **Figura 1** (tomada de jPOS Project Guide) muestra este proceso.

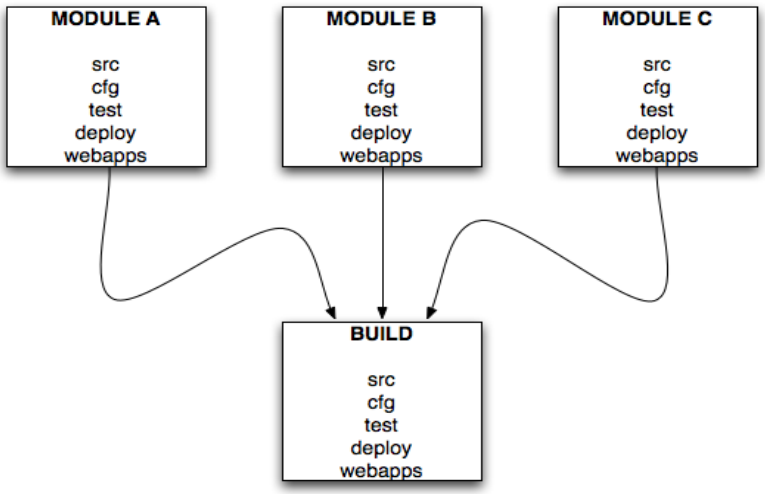

Figura 1 Modelo de integración de módulos de jPOS

jPOS permite ser utilizado como una biblioteca de recursos que un desarrollador puede utilizar en su software. Además provee un interesante auto-contenedor -denominado Q2- que puede ejecutarlo como un servidor "standalone" y con numerosas funcionalidades que se combinan con la metodología de construcción descripta anteriormente.

1. Cada modulo tiene una carpeta "deploy" que contiene archivos xml descriptores de "Qbeans", servicios que ejecuta el Q2.

2. Cada descriptor xml de la carpeta deploy define la clase que lo ejecuta (que debe implementar una cierta interface). Dichas clases se cargan dinámicamente.

3. Provee “hot deploy” de clases java a través de bibliotecas localizadas en la carpeta deploy/lib . Esto permite que se agreguen clases nuevas en tiempo de ejecución. Sumado al punto anterior, se puede agregar prácticamente cualquier funcionalidad en caliente. Esto es de extrema importancia en sistemas de misión crítica porque permite extender funcionalidad y arreglar errores totalmente en línea.

4. Provee un sistema de logs (basados en xml o de formato configurable a través de plantillas) muy completos y totalmente configurables en línea e integrables a los demás Qbeans.

Estas características le dan a jPOS una gran flexibilidad para realizar todo tipo de sistemas transaccionales de misión crítica.

2.2. La comunidad

El proyecto jPOS nació impulsado por Alejandro Revilla alrededor del año 2000 en un proyecto particular. Luego de un tiempo de evolución y un proceso de generalización, jPOS se transformó en un producto OpenSource altamente configurable y adaptable y varios de los que fueron originalmente usuarios se transformaron en colaboradores del proyecto a medida que sus proyectos particulares tenían más y más requerimientos. Hoy en día, este proceso se mantiene de la misma forma. Hay una gran comunidad de programadores que utilizan jPOS en todo el mundo y muchos de ellos comparten sus desarrollos privados con el resto de la comunidad para incrementar la funcionalidad provista por jPOS y cubrir así un espectro cada vez más amplio de negocios.

Hoy por hoy la comunidad debate acerca de nuevas funcionalidades a través de listas de mails y los diferentes módulos se mantienen en Sourceforge, Googlecode y GitHub.

2.3. Las limitaciones

Al ser jPOS un producto pensado para las transacciones electrónicas, la mayor parte de los módulos y las funciones provistas están orientadas al desarrollo transaccional. Desde hace ya un tiempo, se han integrado algunos módulos con la intención de tener interfaces visuales para el monitoreo de las transacciones pero no hay un módulo en la comunidad pensado exclusivamente para generar interfaces administrativas para los modelos de negocios vinculados a las transacciones y para un monitoreo integrado del sistema transaccional.

Este punto constituye uno de los principales problemas a la hora de encarar un proyecto basado en jPOS ya que tiene infinidad de herramientas para las transacciones pero para las interfaces administrativas hay que disparar proyectos paralelos al desarrollo transaccional y esto normalmente insume mucho tiempo y esfuerzo.

Capitulo 3. jPOS Presentation Manager: el desarrollo

3.1. Motivación

Presentado el problema de las interfaces administrativas para los modelos de negocios circunscritos a los sistemas de transacciones electrónicas financieras, junto con las limitaciones actuales de jPOS para abordar dicho problema y sumado a necesidades personales en lo laboral se llega a una propuesta de solución: el jPOS Presentation Manager (jPOS-PM).

El jPOS-PM es un conjunto de módulos integrables a jPOS que tienen como objetivo principal reducir el costo y esfuerzo del desarrollo de una interfaz administrativa para los modelos de negocios basados en transacciones electrónicas financieras realizadas con jPOS, incluyendo a su vez, interfaces visuales para el monitoreo de dichas transacciones. Con el fin de conseguir este objetivo, jPOS-PM ofrece una amplia gama de herramientas para el desarrollador que le permite la generación dinámica de interfaces administrativas y de monitoreo basadas en la misma metodología de desarrollo en la que se sustenta jPOS.

Éste capítulo muestra el proceso de desarrollo llevado a cambo a lo largo del proyecto.

3.2. Análisis de tecnologías

3.2.1. Java

Java[3] es el lenguaje sobre el cual está desarrollado jPOS y por ende, el lenguaje principal para el desarrollo de jPOS-PM. Una clara descripción del lenguaje como parte de una plataforma de desarrollo puede encontrarse en Wikipedia[4]. Describir el uso de Java no es el objetivo de éste informe por lo que no se entrará en detalles.

Si bien el lenguaje resulta heredado, de todos modos hubiese resultado elegido para el desarrollo por su notable flexibilidad, su amplia difusión, su portabilidad y por experiencia previa en su uso.

3.2.2. Ant

Apache Ant [5] es una herramienta usada en programación para la realización de tareas mecánicas y repetitivas, normalmente durante la compilación y construcción (build). Es similar a Make pero sin las engorrosas dependencias del sistema operativo.

Esta herramienta, hecha en Java tiene la ventaja de no depender de las órdenes de Intérprete de comandos / shell de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas, siendo idónea como solución multi-plataforma.

Puntualmente, jPOS utiliza ant para unificar sus módulos en una sola carpeta de output (build) y compilar los fuentes Java. A su vez realiza otras tareas como unificar algunos archivos especiales, como por ejemplo fragmentos de archivos hibernate. Cada módulo puede tener un archivo ant propio y el build.xml general los ejecuta.

3.2.3. Hibernate 3

Hibernate[6] es una herramienta de mapeo objeto-relacional para Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones que permiten establecer estas relaciones.

Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

Hay infinidad de sitios y libros relacionados con Hibernate, incluyendo una definición de Wikipedia [7] y libros tales como *Hibernate in Action[18]* entre otros.

jPOS basa su conexión con la base de datos en hibernate por lo cual se lo utilizará, en caso de ser necesario, para dicho efecto.

jPOS resuelve la construcción del archivo de configuración hibernate.hbm.xml a través del uso de ant, tomando todos los archivos localizados en la carpeta **cfg** de cada módulo que tengan como nombre _**hibernate.cfg.mappings**.

3.2.4. HTML y CSS

La primera decisión a tomar en el desarrollo de jPOS-PM fue acerca de la visualización. Las opciones eran un entorno Web o un entorno de escritorio. La realidad indica que la tendencia de este tipo de sistemas administrativos lleva hacia una interfaz Web dado su entorno distribuido y su amplia difusión.

Sin embargo, a pesar de elegir el entorno Web como visualización principal, es parte del objetivo del proyecto tener bases “independientes” de la vista.

Retomando el objetivo de la sección, se investigaron HTML y hojas de estilo (CSS) con el fin de cumplir lo mejor posible con los estándares w3 y con el objetivo de ofrecer plantillas de diseño para poder adaptar la visualización a los requerimientos del cliente final.

3.2.5. Jetty

Jetty[8] es un servidor HTTP y un contenedor de Servlets desarrollado en Java, que se publica como un proyecto de software libre bajo la licencia Apache 2.0. Debido a su pequeño tamaño, Jetty es ideal para ofrecer servicios Web en una aplicación Java embebida.

Para jPOS este es el servidor idóneo para embeber como un módulo más y, de echo, ya existe en los repositorios de módulos opcionales de jPOS.

Para utilizar el módulo Jetty y montar una aplicación Web en un módulo del jPOS se debe definir una estructura de carpetas y archivos específica que puede apreciarse en la **Figura 2** .

```
jposee
  +->modules
       +->un_modulo
           +->cfg
           |   +->contexts
           |        +->nombre.xml
           |
           +->webapps
           |   +->nombre
           |        +->WEB-INF
           |        |    +->web.xml
           |        |    +-> ...
           |        +-> ...
           +-> ...
```

Figura 2 Estructura de carpetas Jetty

El archivo **nombre.xml** es un archivo de configuración de contexto del Jetty que indica una nueva aplicación Web o contexto dentro del Jetty. Dicho archivo tiene la estructura ilustrada a continuación.

```
<?xml version="1.0"  encoding="ISO-8859-1"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
          "http://jetty.mortbay.org/configure.dtd">
<Configure class="org.mortbay.jetty.webapp.WebAppContext">
  <Set name="contextPath">/nombre</Set>
  <Set name="resourceBase">
     <SystemProperty name="jetty.home" default="."/>
     /webapps/nombre
  </Set>
</Configure>
```

La variable contextPath indica la url relativa desde la cual se accederá a través de un explorador. Por ejemplo, en el caso de la figura sería :

```
http://localhost:8080/nombre
```

asumiendo un entorno local.

La variable resourceBase indica la carpeta donde se encuentra la aplicación Web en cuestión.

Si bien Jetty no es tan usado como Tomcat como contenedor Web, tiene algunos factores

fundamentales a favor que inclinaron la balanza en la decisión:

1. Jetty es mucho más liviano que Tomcat (tanto en tamaño como en consumo de recursos) y por ende es más práctico para embeber en otra aplicación.

2. Jetty es mucho más fácil de embeber en otras aplicaciones (cosa que con Tomcat se puede hacer pero no es fácil)

3. jPOS posee entre sus módulos públicos optativos un servidor Jetty listo para descargar y usar, incluso con SSL disponible y todo configurado para funcionar con jPOS integrado perfectamente.

Todos estos factores (especialmente el último) hicieron que se optara por utilizar Jetty en lugar de Tomcat o cualquier otro contenedor.

3.2.6. Servlets y JSP

Los servlets son objetos que corren dentro del contexto de un contenedor de servlets (como Tomcat o Jetty) y extienden su funcionalidad. Normalmente se utilizan para generar páginas web dinámicamente.

JavaServer Pages (JSP) es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo.

Siendo ambas tecnologías basadas en Java y ampliamente difundidas, se utilizarán para la construcción dinámica de las páginas del jPOS-PM.

3.2.7. Struts

Struts es un framework de desarrollo de aplicaciones Web basado en el patrón MVC utilizado en la plataforma J2EE. El patrón Modelo-Vista-Controlador (MVC) se utiliza ampliamente y es considerado de gran solidez. De acuerdo con este patrón, el procesamiento se separa en tres secciones diferenciadas llamadas el modelo, las vistas y el controlador.

Existen dos versiones de Struts, Struts 1 y Struts 2. Éste proyecto utilizará en primera instancia Struts 1 principalmente por conocimientos previos pero no se descarta la migración o extensión en un trabajo futuro.

3.3. Proceso de ingeniería

> *"Para resolver los problemas reales de una industria, un ingeniero del software o un equipo de ingenieros debe incorporar una estrategia de desarrollo que acompañe al proceso, métodos y capas de herramientas (...) y las fases genéricas (...). Esta estrategia a menudo se llama modelo de proceso o paradigma de ingeniería del software.". [20]*

Éste proyecto de software deberá incorporar su propia estrategia de desarrollo, teniendo en cuenta que el cliente final no es una empresa o persona que compra un trabajo sino que es un grupo específico de usuarios desarrolladores. Sin embargo, el proceso de ingeniería de software es similar al de otros sistemas de información y por lo tanto, se utilizará como base el modelo planteado por Pressman.

El contexto en el que se aplicará el proyecto es un ambiente de desarrollo OpenSource, en el que los usuarios finales o "clientes" son programadores, divididos fundamentalmente en dos perfiles:

1. **Colaborador**: es aquel que además de usar el producto, colabora activamente en la revisión y el desarrollo de nuevas funcionalidades.

2. **Usuario pasivo:** es aquel que utiliza el producto en sus versiones estables y finales sin participar del proceso de desarrollo, validación o prueba.

Un usuario colaborador es, normalmente, un entusiasta del proyecto que espera con ansias nuevas funcionalidades para probar, utilizar y aportar nuevas ideas o módulos. Esto hace que estén habituados a entregas frecuentes, nuevas funcionalidades a diario y versiones experimentales.

Luego de investigar las diferentes metodologías (o paradigmas de la ingeniería de software) para utilizar y teniendo presente el perfil del usuario final, se optó por el modelo incremental.

> *"El modelo incremental combina elementos del modelo lineal secuencial (aplicados repetidamente) con la filosofía interactiva de construcción de prototipos. (...), el modelo incremental aplica secuencias lineales de forma escalonada mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un «incremento» del software (...). Se debería tener en cuenta que el flujo del proceso de cualquier incremento puede incorporar el paradigma de construcción de prototipos."[21]*

Éste modelo de proceso ofrece lineamientos bastante ajustados al perfil de usuario propuesto como destinatario. Cada incremento generado puede tener prototipos que pueden ser evaluados, probados, corregidos y mejorados por colaboradores previamente a cada nueva versión. Una vez finalizada esta etapa de revisión y aceptada la o las funcionalidades nuevas, se pone a disposición de todos el nuevo incremento. De esta forma se satisface las necesidades de todos los perfiles de usuarios.

El gráfico de la **Figura 3** , concreta el modelo de proceso adoptado para éste proyecto.

Figura 3 Proceso de desarrollo de software

En líneas generales, la mayor parte de la funcionalidades empiezan como una **idea**. Puede haber requerimientos específicos pero el usuario final no es un cliente que compra sino que es un programador que opina y por lo tanto las ideas pueden ser tomadas o no.

Luego de tener o recibir una idea, se pasa a la etapa de **análisis** para determinar claramente el objetivo y los costos y beneficios que podría llegar a tener la aplicación.

Finalizado el análisis se procede al **diseño** y a la **codificación** de la o las funcionalidades propuestas. Esta etapa puede realizarse en conjunto con miembros de la comunidad. Como resultado se obtiene un **prototipo** que puede ponerse a disposición de otros miembros de la comunidad para su utilización preliminar y poder realizar una **revisión** y, eventualmente, modificaciones y ajustes.

Para finalizar, se realizan **pruebas** completas modulares e integrales para garantizar el correcto funcionamiento de la nueva funcionalidad y del producto completo y se libera la público una nueva versión o **incremento**.

3.3.1. Análisis de requisitos

Teniendo presente que el usuario final es un desarrollador, los requisitos han sido definidos en base a experiencia y a necesidades surgidas en los diversos proyectos, no a través de un proceso formal de recopilación.

A continuación se presenta la lista de las funcionalidades mínimas requeridas.

1. Se requiere que el módulo principal sea independiente de la visualización final.

2. Se requiere que el módulo principal sea independiente de una base de datos o de cualquier otro origen de datos.

3. Se requiere que puedan se aplicables plantillas de estilos para la personalización de la visualización.

4. Se requiere que pueda existir autenticación de usuario o no (optativamente).

5. Se requiere que la seguridad, de existir, sea independiente de una base de datos o de cualquier otro origen de datos.

6. Se requiere que todos los textos (incluyendo los definidos por el usuario) sean internacionalizables.

7. Se requiere de una construcción configurable, simple, arbitraria y susceptible de aplicación de permisos de menús.

8. Se requiere que los menús puedan tener referencias tanto internas como externas o cualquier otra que el usuario intente definir.

9. Se requiere que el usuario pueda definir entidades de representación visual de clases de su modelo de negocio, configurables a través de xml.

10. Se requiere que cada entidad pueda definir las operaciones que se le podrán aplicar (alta, baja, modificación, consulta, etc).

11. Se requiere que cada entidad pueda representar visualmente sus propiedades internas (campos) de acuerdo a la definición del usuario y a la operación que se esté aplicando.

12. Se requieren las siguientes operaciones predefinidas mínimas

    1. Listado de instancias de una entidad

    2. Ordenamiento de la lista

    3. Filtrado de la lista

    4. Alta de una instancia

    5. Eliminación de una instancia

    6. Modificación de una instancia

    7. Vista de una instancia

13. Se requiere que cualquier operación pueda tener una confirmación antes de ser ejecutada.

14. Se requiere que el usuario pueda definir sus propias operaciones además de las predefinidas por el sistema.

15. Se requiere que el usuario pueda definir o intercalar su propio código en las distintas etapas

de la ejecución de una operación de forma configurable.

16. Se requiere que el usuario pueda resaltar instancias específicas de una entidad bajo algún criterio establecido basado en el valor de alguno de sus campos.

17. Se requiere que se pueda establecer una relación de herencia de campos entre entidades.

18. Se requiere que se puedan ordenar los campos de una entidad de forma arbitraria e independiente de su posición en la definición.

19. Se requiere que se puedan establecer filtros predefinidos a una entidad.

20. Se requiere que se puedan aplicar validaciones a la ejecución de las operaciones aplicadas a una entidad.

21. Se requiere que se puedan aplicar validaciones a cada campo de una entidad, independientemente de la operación en la que se ejecute.

22. Se requiere que puedan aplicar una relación de dependencia "fuerte vs débil" entre entidades, haciendo que las entidades débiles solamente puedan existir relacionadas con una fuerte.

23. Se requiere la persistencia a lo largo de una sesión de usuario de los filtros, el ordenamiento y otras opciones aplicables a una entidad particular.

24. Se requiere que se puedan monitorizar en línea archivos, tablas de base de datos o cualquier otra variable propensa a monitoreo.

3.3.2. Diseño: arquitectura de software

Como diseño central se tomará la definición de la arquitectura del sistema. Como describe la siguiente cita de D. Garlan y M. Shaw, la arquitectura de software es lo que define las interrelaciones entre los módulos del sistema a desarrollar, los framework y patrones a adoptar dependiendo del ámbito subyacente al problema a resolver, más allá del lenguaje de programación a adoptar.

> *"As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.*
>
> *This is the software architecture level of design. There is a considerable body of work on this topic, including module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component*

integration mechanisms. In addition, an implicit body of work exists in the form of descriptive terms used informally to describe systems".[19]

Para abordar el diseño de la arquitectura se utilizará como lenguaje de representación visual UML[9]

3.3.2.1. Diseño modular

El diseño modular representa la estructura, composición e interrelación de los módulos jPOS a desarrollar.

El objetivo no es sólo ofrecer un módulo que contenga toda la funcionalidad sino que se busca desarrollar un módulo "base" independiente de la visualización final para el usuario[3] y que sirva como punto de partida para otros módulos específicos que usuarios de la comunidad quieran aportar.

Junto con el módulo base se proveerá un conjunto de módulos que darán una implementación particular lista para usar.

La **Figura 4** muestra los módulos a definir y su interacción. Se utiliza la representación en UML con diagramas de despliegue.

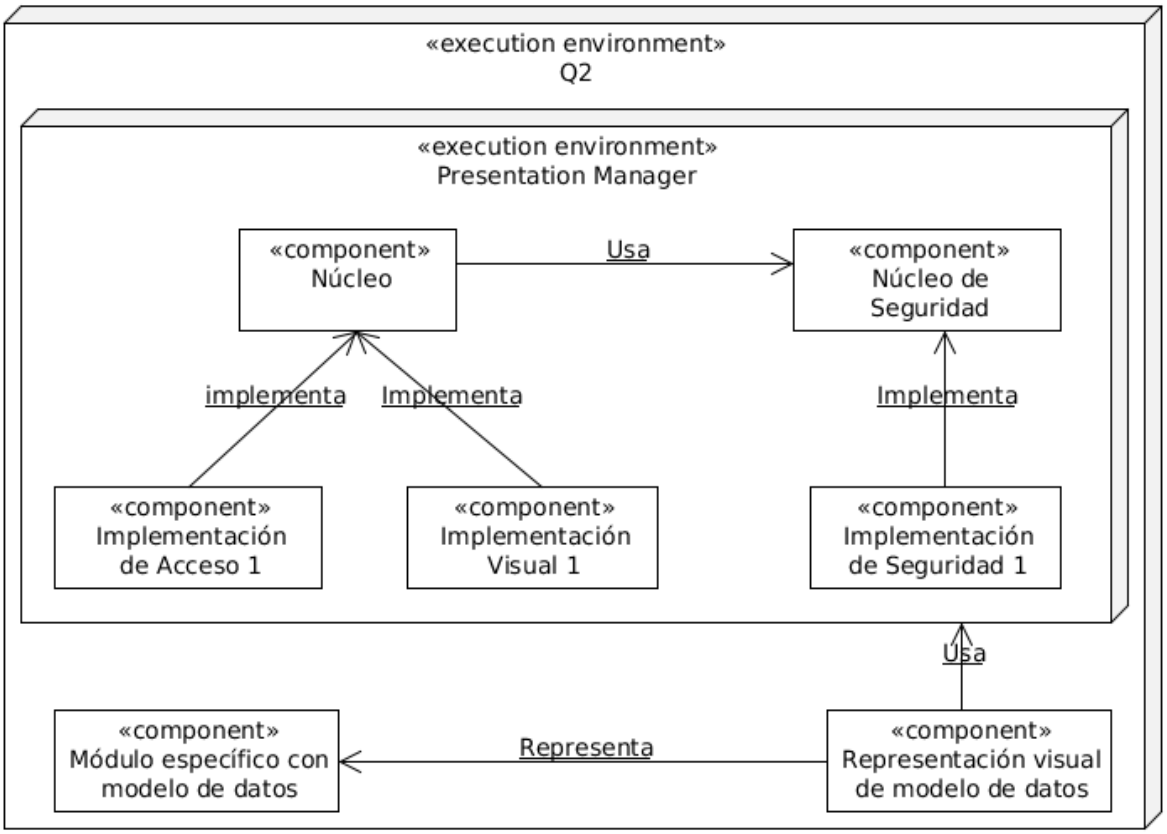

Figura 4 Diseño modular: diagrama de despliegue

3 Web con struts, Web con servlets, standalone con Swing, etc

- Cada modulo está contenido en el ambiente del contenedor jPOS Q2.

- Cada módulo del jPOS-PM está contenido simbólicamente en un ambiente de ejecución específico.

- Existe un núcleo fundamental que provee una abstracción independiente de la visualización final e independiente del origen de los datos. Éste módulo se denominará “pm_core”.

- Existe un núcleo central para abstraer la seguridad y que es usado por el pm_core. Éste módulo se denominará “pm_security_core”.

- En un ambiente normal, se selecciona y aplica un módulo específico que implementa el acceso u origen de los datos. En caso de necesitarse conexión con la base de datos, se utilizará un módulo denominado “pm_core_db”.

- En un ambiente normal, se selecciona y aplica un módulo específico que implementa la visualización final de las interfaces. En éste trabajo se realizará una implementación en Web con Struts y se denominará a dicho módulo “pm_struts”.

- En un ambiente normal, se selecciona y aplica un módulo específico que implementa la seguridad. En caso de utilizarse seguridad basada en la base de datos, se utilizará un módulo denominado “pm_security_db”.

- Por último, el usuario posee su modelo de datos en un módulo particular e implementa, utilizando el jPOS-PM, un módulo de representación visual de su modelo.

3.3.2.2. Diseño de clases

El diseño de clases ha sido dividido en submódulos funcionales para mayor claridad de lectura y análisis.

i) Modelo de clases central

Las clases principales del módulo pm_core abarcan la definición completa de la representación visual de entidades, campos, operaciones, resaltados y la abstracción del origen de datos. Puede verse el modelo en la **Figura 5** .

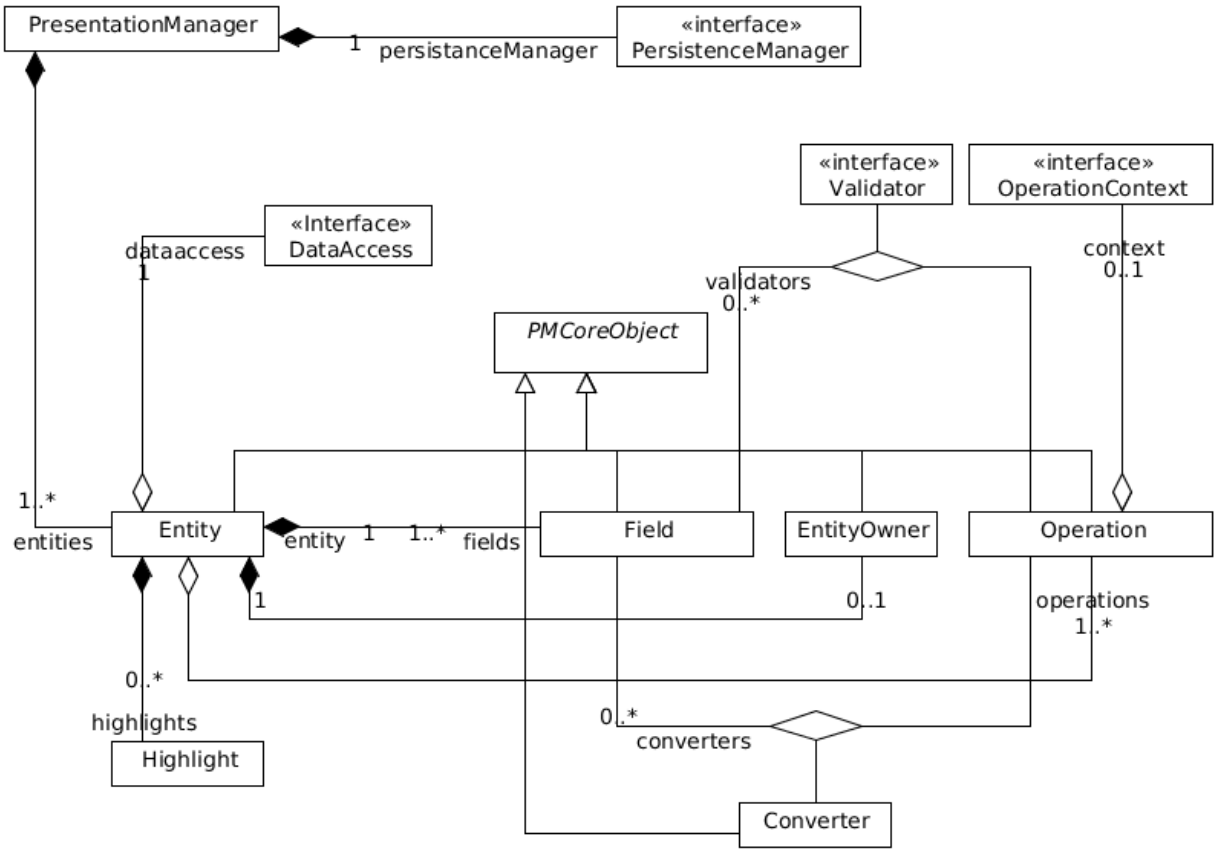

Figura 5 Diagrama de clases central

Las entidades presentes en el diagrama de clases se describen en la **Tabla 1.**

Clases e Interfaces	
PresentationManager	Singleton que contiene métodos de ayuda, funciona como contenedor de todas las entidades y monitores y contiene toda la configuración

Clases e Interfaces	
	general.
PMCoreObject	Superclase central. Provee métodos de ayuda generales.
Entity	Representación visual de un objeto del modelo de negocios. Contiene sus campos, operaciones, resaltados y la metodología de acceso a los datos.
Field	Representación visual de un campo de un objeto del modelo de negocio.
EntityOwner	Identificación de la entidad padre de una entidad débil.
Operation	Representación de una operación que puede realizarse a una entidad, por ejemplo, listar, alta, baja, etc.
Highlight	Representación de una condición para resaltar en ciertas operaciones algunas instancias de una entidad.
Converter	Forma en la que un field (campo) se visualizará en una determinada operación.
Validator	Interface para validar que se cumplan ciertas condiciones luego de aplicar el converter en una operación.
OperationContext	Interface que permite ejecutar distintos métodos en tres instancias de la ejecución de una operación: previamente a la conversión, posterior a la conversión pero previo al salvado final y luego del salvado.
PersistanceManager	Interface para contextualizar y abstraer el proceso transaccional (startTransaction, commit, abort, etc). Necesita un rediseño.
DataAccess	Interface para determinar el origen y el destino de los datos.

Tabla 1. Descripción de entidades y relaciones del modelo de clases principal

Nota: PersistanceManager y DataAccess están muy relacionados entre sí ya que normalmente el acceso a los datos requiere de un contexto transaccional particular. Por ejemplo, para salvar un objeto a una base de datos (utilizando el DataAccess) se debe estar en un contexto de transacción de la base de datos, el cual es brindado por el PersistenceManager. Sin embargo se puede ver en el modelo que todo el sistema tiene sólo un PersistanceManager pero cada entidad puede tener su DataAccess. Este es un problema de diseño que está pendiente para una resolución futura ya que a nivel de implementación presenta varios problemas para resolver. El ideal sería que cada DataAcces tenga su propio PersistanceManager.

ii) Modelo de clases de monitores

El módulo pm_core provee un pequeño submodelo para representar monitores. Un monitor tiene como objetivo visualizar en línea las variaciones de un cierto origen de datos, ya sea un archivo, una tabla, un objeto, etc. El diseño está basado en el patrón Observer[10]. Se puede ver el diagrama en la **Figura 6** .

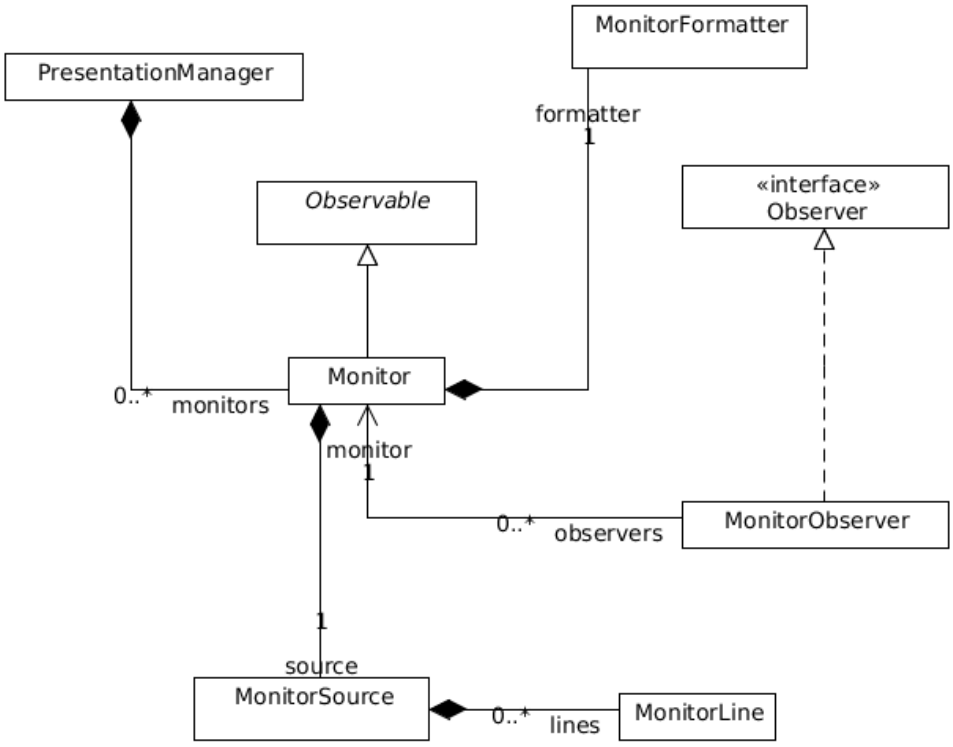

Figura 6 Diagrama de clases de monitores

Clases e Interfaces	
Monitor	Objeto observable que mantiene el estado de lo que se está monitoreando.
MonitorLine	Representa una línea del monitor.
MonitorSource	Representa el origen de los datos del monitor.
MonitorFormatter	Es el encargado de darle formato visual a cada línea del monitor.
MonitorObserver	Observador del monitor.

Tabla 2. Descripción de entidades y relaciones del modelo de clases de monitores

El monitor mantiene un conjunto de líneas de datos (refrescados a intervalos regulares configurables) en la medida en que tenga observadores. Los observadores son informados de los cambios con una lista de strings formateados (por medio del formatter).

iii) Modelo de clases de menús

El módulo pm_core posee un pequeño submódulo funcional para la representación de los menús que normalmente tiene cualquier sistema administrativo. Para la implementación de este submódulo se utilizó el patrón de diseño composite[11]. Se puede ver el diagrama en la **Figura 8** .

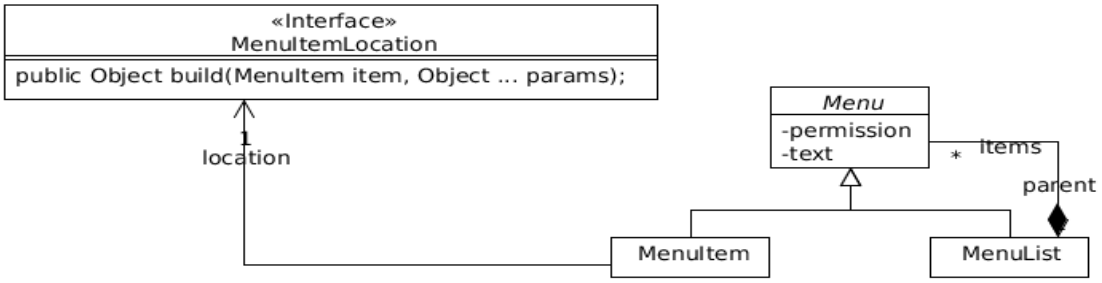

Figura 7 Diagrama de clases de menús.

Clases e Interfaces	
Menu	Superclase abstracta representando un menú arbitrario.
MenuItem	Hoja del árbol de menús. Éste es el item que posee la referencia al lugar donde deriva esta entrada de menú.
MenuList	Es un nodo intermedio en el árbol de menús. Posee una lista arbitraria de otros menús.
MenuItemLocation	Representa el destino de un ítem, el "donde" derivará el acceder a dicha opción de menú.

Tabla 3. Descripción de entidades y relaciones del modelo de clases de menús

La idea detrás del "location" es abstraer la representación visual de las "pantallas". Por ejemplo, una implementación Web tendrá un location como un link (), una implementación Swing tendrá un location que acceda a un formulario, etc.

Actualmente, un nodo del árbol (MenuList) no tiene una referencia a un destino, pero en un futuro podría tenerlo.

iv) Modelo de clases de Seguridad

El modelo de seguridad está incluido en el módulo pm_security_core y representa los usuarios con sus permisos, grupos y perfiles junto con el mecanismo de generalización provisto para abstraer el origen de los datos. Se puede ver el diagrama en la **Figura 8** .

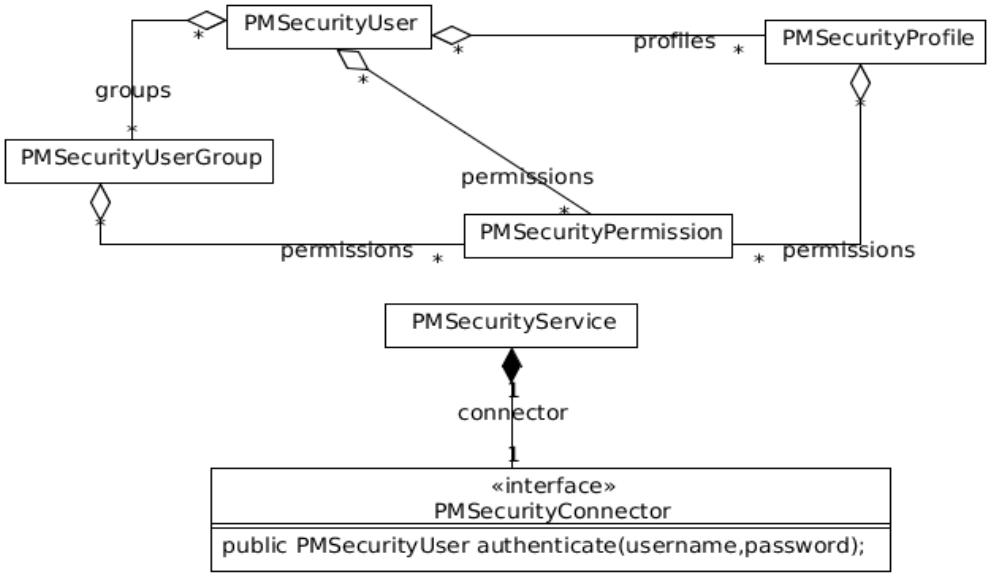

Figura 8 Diagrama de clases de seguridad

Clases e Interfaces	
PMSecurityUser	Representa a un usuario del sistema.
PMSecurityPermission	Representa a un permiso para realizar una o más acciones específicas dentro del sistema.
PMSecurityUserGroup	Representa un grupo de usuarios que tiene un conjunto específico de permisos.
PMSevurityProfile	Representa un perfil de usuario que tiene un conjunto específico de permisos.
PMSecurityService	Qbean encargado de administrar la seguridad
PMSecurityConnector	Interface que abstrae el comportamiento general de la seguridad.

Tabla 4. Descripción de entidades y relaciones del modelo de clases de seguridad

A través de la interface connector (simplificada en la imagen) se definen todos los métodos necesarios para interactuar con cualquier seguridad que el usuario defina. Los métodos incluyen la autenticación, cambio de contraseña, alta, baja, modificación y listado de usuarios, grupos, perfiles y permisos.

Cualquier módulo específico que determine un origen de datos para estas clases deberá, a través de un conector, ofrecer estas operaciones e traducir su representación interna en éste modelo.

3.3.2.3. Diseño de API

El usuario final de éste sistema será un desarrollador y por lo tanto, se debe ofrecer una API para que pueda usar. Como parte del objetivo se planteó respetar la metodología de desarrollo de jPOS y la misma se basa en definición a través de archivos de configuración xml.

A continuación se presentan las definiciones de los archivos DTD[12] de cada ítem configuración ofrecida al desarrollador. En el capítulo siguiente se detallará con más precisión el contenido de cada elemento y su significado.

i) DTD de archivos de entidades

Cada entidad que representa a una clase del modelo de negocio se configura a través de un archivo xml que luego es incluido en el archivo de configuración del Qbean jPOS del jPOS-PM.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % xhtml1-lat1 SYSTEM "xhtml1-lat1.ent">  %xhtml1-lat1;
<!ELEMENT entity ( id, clazz, listfilter?, owner?, auditable, persistent,
operations, rowsPerPage?, field+ ) >
<!ELEMENT id (#PCDATA) >
<!ELEMENT clazz (#PCDATA) >

<!ATTLIST entity
    id CDATA #REQUIRED
    clazz CDATA #REQUIRED >

<!ELEMENT listfilter (EMPTY) >
<!ATTLIST listfilter class
    CDATA #REQUIRED>

<!ELEMENT rowsPerPage (#PCDATA) >
<!ELEMENT auditable (#PCDATA) >
<!ELEMENT persistent (#PCDATA) >
<!ELEMENT owner (entity_id , entity_property, local_property?) >

<!ELEMENT entity_id (#PCDATA) >
<!ELEMENT entity_property (#PCDATA) >
<!ELEMENT local_property (#PCDATA) >

<!ELEMENT operations (operation*) >
<!ELEMENT operation (id, enabled?, scope?, visibleIn?, url?, context?,
validators?, properties?) >

<!ATTLIST operation
    id CDATA #REQUIRED
    enabled CDATA #IMPLIED
    scope CDATA #IMPLIED
    visibleIn CDATA #IMPLIED >

<!ELEMENT context ( EMPTY ) >
<!ATTLIST context class
    CDATA #REQUIRED>

<!ELEMENT text (#PCDATA) >
<!ELEMENT icon (#PCDATA) >
<!ELEMENT enabled (#PCDATA) >
```

```
<!ELEMENT scope (#PCDATA) >
<!ELEMENT visibleIn (#PCDATA) >
<!ELEMENT url (#PCDATA) >

<!ELEMENT validators (validator*) >
<!ELEMENT validator (#PCDATA) >

<!ELEMENT field (id, property?, display?, width?, align?, searchCriteria?,
multiEditable?, converters?, field-validator*) >

<!ATTLIST field
    id CDATA #REQUIRED
    property CDATA #IMPLIED
    display CDATA #IMPLIED
    width CDATA #IMPLIED
    align CDATA #IMPLIED>

<!ELEMENT display (#PCDATA) >
<!ELEMENT property (#PCDATA) >
<!ELEMENT width (#PCDATA)>
<!ELEMENT converters (converter*)>
<!ELEMENT converter (properties?)>
<!ATTLIST converter
    class   CDATA #REQUIRED
    operations  CDATA #REQUIRED >
<!ELEMENT field-validator (properties?)>
<!ELEMENT properties (property*)>
<!ELEMENT property ( EMPTY )>
<!ATTLIST property
    name    CDATA #REQUIRED
    value   CDATA #REQUIRED >
<!ELEMENT listfilter (#PCDATA)>
<!ELEMENT showToList (#PCDATA)>
<!ELEMENT align (#PCDATA)>
<!ELEMENT searchCriteria (#PCDATA)>
```

ii) DTD de archivo de Locations

El archivo de locations define las locations disponibles para utilizar en el menú.

El archivo utilizado debe llamarse “pm.locations.xml” y localizarse en la carpeta cfg del módulo de interfaz.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % xhtml1-lat1 SYSTEM "xhtml1-lat1.ent"> %xhtml1-lat1;
<!ELEMENT locations (location*)>
<!ELEMENT location (EMPTY)>
<!ATTLIST location id CDATA #REQUIRED>
<!ATTLIST location class CDATA #REQUIRED>
```

iii) DTD de archivo de menús

El archivo de menú define la estructura del menú con su anidamiento. El menú base puede contener tanto ítems como listas y cada ítem debe contener su location que define “donde” llevará el ingresar a dicho opción de menú.

El archivo utilizado debe llamarse “pm.menu.xml” y localizarse en la carpeta cfg del módulo de interfaz.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % xhtml1-lat1 SYSTEM "xhtml1-lat1.ent">%xhtml1-lat1;
<!ELEMENT menu (menu-item*, menu-list*)>

<!ELEMENT menu-item (location)>
<!ATTLIST menu-item
    text CDATA #REQUIRED
    perm CDATA #REQUIRED>

<!ELEMENT location (EMPTY)>
<!ATTLIST location
    id CDATA #REQUIRED
    value CDATA #REQUIRED>

<!ELEMENT menu-list (menu-list* , menu-item*)>
<!ATTLIST menu-list
    text CDATA #REQUIRED
    perm CDATA #REQUIRED>
```

iv) DTD de archivo de monitores

Al igual que una entidad, cada monitor se define en un archivo de configuración propio y luego se incluye en el archivo de configuración general del QBean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % xhtml1-lat1 SYSTEM "xhtml1-lat1.ent">  %xhtml1-lat1;
<!ELEMENT monitor ( id, source, formatter, delay, max?, cleanup?, all? ) >
<!ELEMENT id (#PCDATA) >

<!ELEMENT formatter (EMPTY) >
<!ATTLIST formatter class CDATA #REQUIRED>

<!ELEMENT source (properties?)>
<!ELEMENT properties (property*)>
<!ELEMENT property ( EMPTY )>
<!ATTLIST property
    name    CDATA #REQUIRED
    value   CDATA #REQUIRED>

<!ELEMENT delay (#PCDATA) >
<!ELEMENT max (#PCDATA) >
<!ELEMENT cleanup (#PCDATA) >
<!ELEMENT all (#PCDATA) >
```

3.3.3. Pruebas

Dado que el sistema es fundamentalmente una API o framework de desarrollo, las pruebas se realizan a través de sistemas de ejemplo que usan la mayor cantidad de variantes y funciones posibles. Uno de dichos sistemas se encuentra alojado en http://github.com/jpaoletti/jPOS-PM-Test y contiene una amplia variedad de pruebas orientadas principalmente a los distintos tipos de entidades y tipos de datos de los campos de las entidades.

Algunas opciones son probadas directamente por usuarios finales en prototipos liberados para la comunidad OpenSource dado que muchas veces son funcionalidades específicas que dependen de un ambiente difícil de reproducir.

3.3.4. Mantenimiento

> *"La fase de mantenimiento se centra en el cambio que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del software y a cambios debidos a las mejoras producidas por los requisitos cambiantes del cliente." (11)*

Al ser un proyecto OpenSource, el primer requisito a satisfacer es la disponibilidad del código fuente para su descarga, modificación, adaptación y uso. Esto, en combinación con la necesidad de mantenimiento y evolución, lleva a que el proyecto deba ser alojado en algún sitio de acceso público de gestión de proyectos. Para dicho fin, se seleccionó GitHub como alojamiento principal.

La dirección del proyecto es

http://github.com/jpaoletti/jPOS-Presentation-Manager

o

http://jpaoletti.github.com/jPOS-Presentation-Manager/

En dicho sitio puede encontrarse

1. El código fuente
2. La información de cada commit
3. Un gestor de incidencias para el reporte y resolución de errores.
4. Una sección de descargas para las versiones estables y para la documentación
5. Una Wiki para documentación y discusión.

Capitulo 4. jPOS Presentation Manager: el producto

El capítulo anterior presentó al jPOS-PM como el resultado de un proceso de ingeniería de software, con su arquitectura y diseño. El objetivo de éste capítulo es presentarlo desde el punto de vista de un producto del mercado, con una descripción de sus mejores funcionalidades y cualidades.

La mayor parte del contenido de éste capítulo se encuentra resumido (y en inglés) en la *jPOS Presentation Manager Quick Guide* (Guía Rápida del jPOS-PM, obtenible desde http://jpaoletti.github.com/jPOS-Presentation-Manager/download.html), documento pensado para ser una referencia rápida para el programador que utilice el sistema.

A lo largo del capítulo, será utilizado como ejemplo para mostrar cada funcionalidad (en los casos que sea posible) el jPOS-PM-Test obtenible desde http://github.com/jpaoletti/jPOS-PM-Test/downloads .

4.1. Instalación y configuración

En este capítulo se presenta la metodología de instalación y configuración del jPOS-PM en el contexto de un proyecto jPOS.

Por el momento y hasta liberar una versión estable, el modo de instalación es a través de la descarga del código fuente desde http://github.com/jpaoletti/jPOS-Presentation-Manager .

4.1.1. Integración a jPOS

Asumiendo que ya se tiene un proyecto jPOS configurado[4], deben agregarse a la carpeta modules los módulos del jPOS-PM que se utilizarán. Como mínimo:

- pm_core
- pm_struts (de momento la única implementación visual)
- pm_security_core

Además, deben estar presentes los siguientes módulos requeridos:

- jetty6
- commons
- jpos
- system
- constants

Por último, lo que normalmente sucede en un proyecto jPOS es que el programador crea un módulo específico para su proyecto. En el caso de jPOS-PM-Test dicho módulo se denomina “pm_test”. Es recomendado que toda la configuración y código fuente generado para jPOS-PM se incluya en un nuevo módulo con el sufijo “ui”. En el caso del test sería “pm_test_ui”. De esta forma quedan perfectamente separados el modelo de datos de la visualización.

El módulo ui debe tener la estructura interna definida por la **Figura 9** :

```
modules
  →  project_ui
     → src
     → deploy
     → cfg
           → entities
           → monitors
     → webapps
           → pm
                 → templates
```

4 Como se especifica en http://jpos.org/doc/jPOS-Project-Guide.pdf

Figura 9 Estructura interna de modulo ui

La **Figura 10** muestra la configuración final del proyecto jPOS-PM-Test

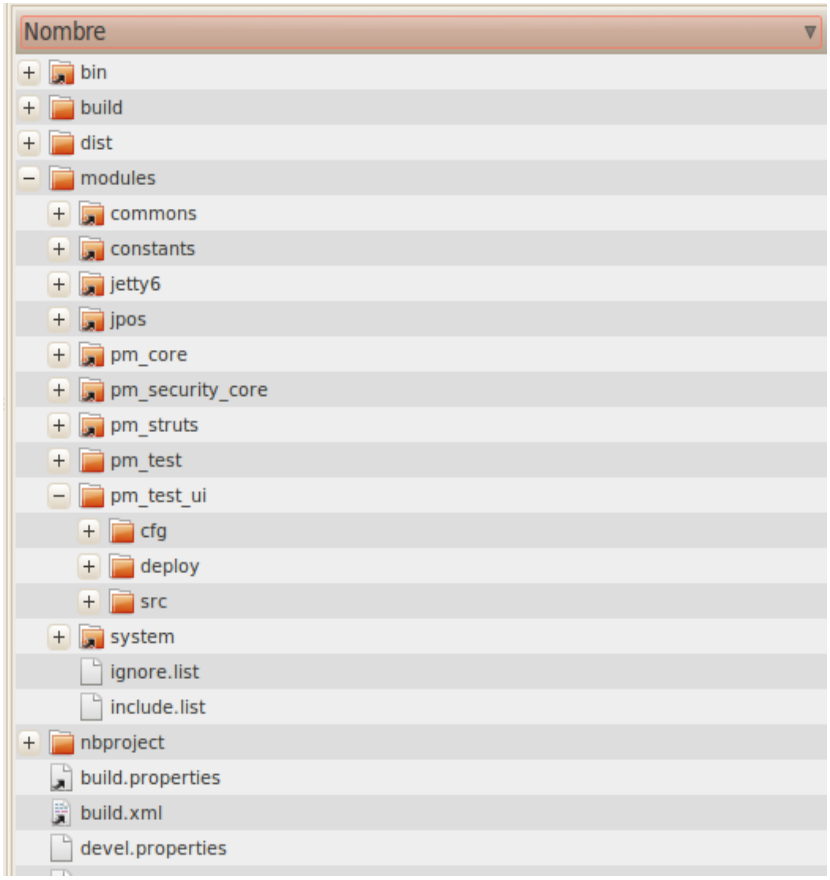

Figura 10 Estructura interna de proyecto jPOS

Si bien la carpeta webapps es opcional y no aparece en este caso, es muy probable que sea necesaria.

4.1.2. Configuración de un proyecto

La configuración de un proyecto, luego de creada la estructura de carpetas apropiadas, se basa en la creación de algunos archivos.

4.1.2.1. Archivo de jetty

Como primer paso debe copiarse el archivo

```
modules/jetty6/cfg/jetty.xml.sample
```

a

```
modules/project_ui/cfg/jetty.xml
```

De esta manera Jetty estará disponible al iniciar jPOS. Dentro de este archivo pueden configurarse todas las opciones del contenedor web, por ejemplo los puertos (http, https), los certificados, etc. Por defecto, los puertos son 8080 (http) y 8443 (https).

4.1.2.2. QBean

El QBean es la parte "ejecutable" de cualquier módulo jPOS y es configurado a través de un archivo xml ubicado en la carpeta **deploy** de un módulo. En este caso, el archivo debe estar localizado en la carpeta deploy del módulo ui y se recomienda el nombre "80_pm.xml".

deploy/80_pm.xml

```
<presentation-manager class="org.jpos.ee.pm.struts.PMStrutsService"
logger="Q2">
    <property name="debug"               value="true | false" />
    <property name="template"            value="default" />
    <property name="appversion"          value="1.0.0" />
    <property name="login-required"      value="true | false" />
    <property name="ignore-db"           value="true | false" />
    <property name="title"               value="main.title" />
    <property name="subtitle"            value="main.subtitle" />
    <property name="default-data-access" value="org.jpos.ee.pm.xxxx"/>
    <property name="persistence-manager" value="org.jpos.ee.pm.yyyy"/>

    <!-- Busissnes Entities -->
    <property name="entity"              value="cfg/entities/xxxx.xml" />
    <!-- ... -->

    <!-- Monitors -->
    <property name="monitor"             value="cfg/monitors/q2.xml" />
    <!-- ... -->
</presentation-manager>
```

Propiedad	Descripción
debug	Activa mensajes de debug internos.
template	Id de la plantilla. La plantilla por defecto es “default” .
appversion	Versión de la aplicación.
login-required	Activa / desactiva la seguridad (y con ello el login).
ignore-db	Cuando es “true” desactiva cualquier acceso a la base de datos.
title	Clave de los archivos de internacionalización para el título.
subtitle	Clave de los archivos de internacionalización para el subtítulo.
default-data-access	Clase del DataAcces por defecto para las entidades que no lo especifiquen.
persistence-manager	Clase del manejador de persistencia.
entity*	Uno o más archivos de entidades.
monitor*	Uno o más archivos de monitores.

Tabla 5. Propiedades del archivo de configuración del qbean de jPOS-PM

4.1.3. Funciones generales

4.1.3.1. Internacionalización

Todos los textos son y deben ser internacionalizables, es decir, propensos a traducción. La forma en la que jPOS-PM resuelve esto es a través de archivos ApplicationResouce contenidos en la carpeta cfg del módulo ui y que tienen la siguiente forma:

```
cfg/ApplicationResource.properties
--------------------------------------------------------------------------
main.title=Some Title
main.subtitle=Some Subtitle
...
```

Para definir las traducciones a los diferentes idiomas se utiliza la nomenclatura

```
cfg/ApplicationResource_xx_YY
```

Donde xx es el idioma basado en la codificación definida por el estándar ISO-639-1 [13] [14] e YY es el país basado en la codificación de dos letras definida por el estándar ISO-3166-1 [15] [16].

4.1.3.2. Templates y diseño gráfico

El jPOS-PM tiene una política de plantillas o templates basada en hojas de estilos. Todos los templates residen en la carpeta webapps/pm/templates y consisten en una carpeta principal (por ejemplo “default”) y dentro de dicha carpeta debe haber, como mínimo, un archivo “all.css” que normalmente importa otros archivos css a través de la sentencia “@import url("xxx.css");”.

Si bien jPOS-PM tiene una plantilla por defecto (“default”), es recomendado copiar dicha plantilla en la carpeta webapps/pm/templates del módulo ui del proyecto (y con otro nombre), ya que de esta manera, se le pueden realizar cualquier tipo de ajustes y agregados necesarios o incluso, cambiar por completo el diseño gráfico.

Dentro de la misma carpeta de cada template hay dos carpetas con las imágenes (img e images) de los botones y de los estilos que las usen. Está planificado para un futuro, unificar estas carpetas.

Para cambiar el template que se está usando, es suficiente cambiando la propiedad “template” en el archivo 80_pm.xml . Esto cambia **en línea** el template utilizado.

```
    <property name="template" value="mytemplate" />
```

4.1.3.3. Seguridad

Para incluir cualquier tipo de seguridad (y por ende, login), se debe definir un QBean que especifique la clase del conector que se utilizará para obtener la información. El módulo pm_security_core define la interface que debe satisfacer un conector y el módulo optativo "pm_security_db" implementa esta interfaz con un modelo simplificado de base de datos.

El archivo de configuración del QBean de seguridad se define como:

```
79_security.xml
--------------------------------------------------------------------------------
<security-manager class="org.jpos.ee.pm.security.core.PMSecurityService">
  <property name="connector" value="org.jpos.ee.pm.security.XxConnector" />
</security-manager>
```

El usar seguridad habilita

1. Login de usuarios
2. Link de perfil de usuario
3. Uso de los atributos "perm" en la definición del menú.

Este esquema "core-implementation" permite tener cualquier implementación de seguridad. Por ejemplo podría desarrollarse un módulo que utilice un archivo de configuración xml o un Web Service o un sistema LDAP para autenticar los usuarios.

También se provee optativamente un módulo "pm_security_ui" que contiene una implementación de la representación visual, basada en el jPOS-PM, del submodelo de seguridad propuesto, y que utiliza el conector para abstraerse del origen de los datos.

A continuación se muestran algunas pantallas de este módulo para dar una idea del contenido del módulo de seguridad.

Figura 11 Menú de módulo de seguridad

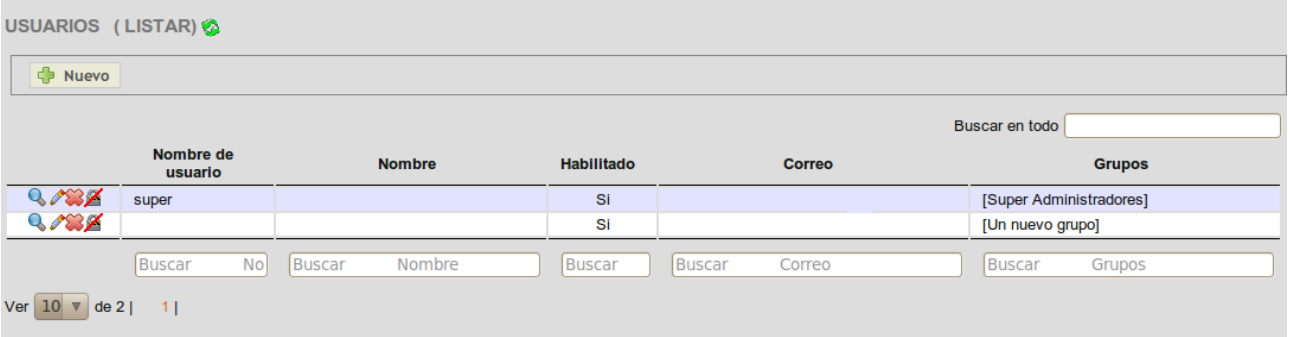

Figura 12 Lista de usuarios

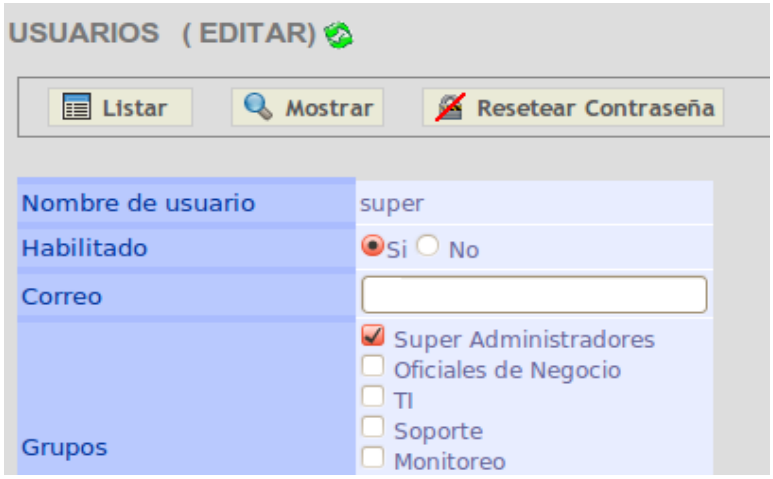

Figura 13 Edición de un usuario

Figura 14 Lista de grupos

4.1.3.4. Acceso a datos

El acceso a los datos de origen de cada entidad es configurable a través de los DataAccess. Un data access es una interface que tiene todas las operaciones básicas que se pueden realizar sobre cualquier entidad: listar, editar, agregar, borrar, refrescar. También tiene un método para crear filtros específicos que utiliza cada DataAccess en particular en la operación listar.

El módulo “pm_core_db” implementa un DataAccessDB que utiliza la clase org.jpos.ee.DB para obtener datos de una base de datos utilizando hibernate.

El módulo “pm_core” provee un DataAccessVoid por defecto que no tiene ningún comportamiento, es decir que las operaciones no “persisten” hacia ningún origen de datos.

Normalmente, cuando se utiliza un DataAccess (en particular el que accede a la base de datos), se necesita un contexto transaccional, es decir, un inicio de transacción, commit y rollback. Estas operaciones las provee de forma abstracta la interface PersistenceManager. A diferencia del DataAccess, el PersistenceManager se define a nivel global (y no particular a la entidad). Esto establece una limitación al sistema que intentará ser resuelta en versiones futuras.

En el jPOS-PM-Test no hay acceso a la base de datos por lo que cada entidad de prueba que allí se presenta, utiliza un DataAccess específico que genera aleatoriamente y luego mantiene en memoria una serie de objetos, simplemente guardando dicha lista en una variable de instancia.

El DataAccess es invocado generalmente al final de la ejecución de una operación. Por ejemplo, luego de editar un objeto, al finalizar la operación se invoca al método “update”.

4.2. API para el desarrollador

En este capítulo se introducen las funciones principales que puede utilizar el desarrollador a lo largo de la creación de la interfaz del proyecto. El objetivo es dar un idea general de las herramientas sin entrar demasiado en detalle. Para más información se puede descargar la guía rápida desde http://jpaoletti.github.com/jPOS-Presentation-Manager/download.html .

4.2.1. Menú

El menú se configura por medio de dos archivos obligatorios, el de "locations" y el de menú propiamente dicho.

4.2.1.1. Archivo de "locations"

Éste archivo debe crearse en la carpeta cfg del módulo ui y debe contener la definición de todas las localizaciones (locations) que podrán usarse luego en el menú.

Cada localización es una clase que implementa la interface MenuItemLocation, específica del módulo de implementación visual (en este ejemplo, pm_struts) y que le dice al menú "como" construir su contenido de modo tal que lleve a otra pantalla del sistema. En este ejemplo, pm_struts tiene dos implementaciones de MenuItemLocation

```
cfg/pm.locations.xml
--------------------------------------------------------------------------------
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE menu SYSTEM "locations.dtd">
<locations>
<location id="pmstruts"
          class="org.jpos.ee.pm.struts.MenuItemLocationStruts"/>
<location id="external"
          class="org.jpos.ee.pm.struts.MenuItemLocationExternal"/>
</locations>
```

pmstruts es el location básico y construye un link html de la forma

```
<a href="javascript:loadPage('xxxx')">yyyy</a>
```

external construye un link html externo

```
<a href="xxxx" target="_blank">yyyy</a>
```

donde "xxxx" es el valor dado en la definición del menú e "yyyy" es el texto del menú.

4.2.1.2. Archivo de menús

Este archivo debe crearse en la carpeta cfg del módulo ui y debe contener la definición de todo el menú del sistema. La forma general del menú es la siguiente:

cfg/pm.menu.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE menu SYSTEM "menu.dtd">
<menu>
    <menu-list text="pm.menu.list.xxxx" perm="">
        <menu-list text="pm.menu.item.xxxx" perm="">
            <menu-item text="pm.menu.item.xxxx" perm="">
                <location id="pmstruts" value="/list.do?pmid=yyyy" />
            </menu-item>
            <menu-item text="pm.menu.item.yyyy" perm="">
                <location id="external" value="http://jpos.org" />
            </menu-item>
        </menu-list>
    </menu-list>
</menu>
```

Los atributos “text” son claves del archivo de internacionalización (ver más adelante). Los atributos “perm” son permisos del módulo de seguridad. Para que un usuario pueda ver dicha opción de menú, debe tener el permiso definido en el archivo de menú.

El jPOS-PM-Test tiene el siguiente archivo de menú.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE menu SYSTEM "menu.dtd">
<menu>
    <menu-item text="pm.menu.item.jpos" perm="">
        <location id="external" value="http://jpos.org" />
    </menu-item>
    <menu-list text="pm.menu.list.monitor" perm="">
        <menu-item text="pm.menu.item.q2log" perm="">
            <location id="pmstruts" value="/monitor.do?pmid=q2" />
        </menu-item>
    </menu-list>

    <menu-list text="pm.menu.list.business" perm="">
        <menu-item text="pm.menu.item.simpleclass" perm="">
            <location id="pmstruts" value="/list.do?pmid=simpleclass" />
        </menu-item>
        <menu-item text="pm.menu.item.complexclass1" perm="">
            <location id="pmstruts" value="/list.do?pmid=complexclass1" />
        </menu-item>
        <menu-item text="pm.menu.item.complexclass2" perm="">
            <location id="pmstruts" value="/list.do?pmid=complexclass2" />
        </menu-item>
        <menu-item text="pm.menu.item.parentclass" perm="">
            <location id="pmstruts" value="/list.do?pmid=parentclass" />
        </menu-item>
    </menu-list>

    <menu-list text="pm.menu.list.main.rose" perm="">
        <menu-list text="pm.menu.list.rose" perm="">
            <menu-list text="pm.menu.list.rose" perm="">
                <menu-list text="pm.menu.list.rose" perm="">
                    <menu-item text="pm.menu.item.pm" perm="">
                                                <location  id="external"
value="http://github.com/jpaoletti/jPOS-Presentation-Manager" />
                    </menu-item>
```

```
                </menu-list>
            </menu-list>
        </menu-list>
    </menu-list>
</menu>
```

Lo que produce el menú que puede verse en la Figura 15 y la Figura 16 (con la plantilla por defecto).

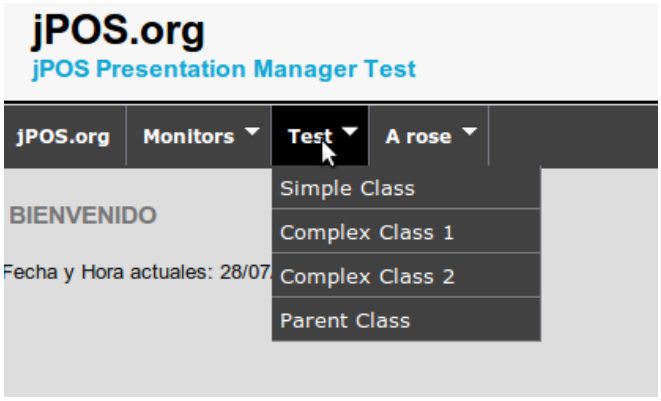

Figura 15 Menú de jPOS-PM-Test. Ejemplo 1

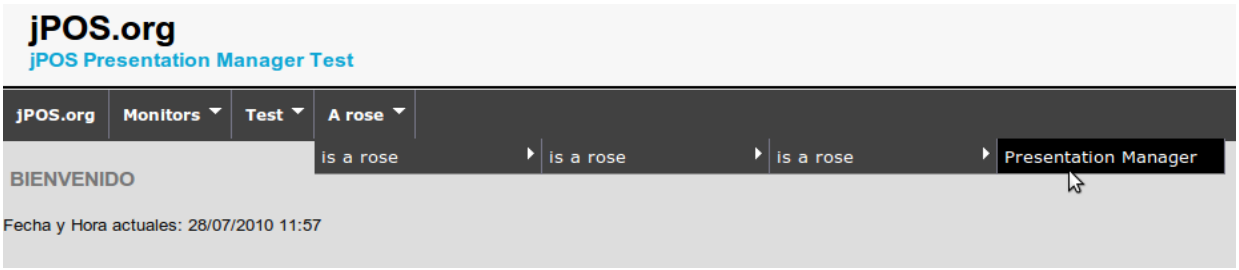

Figura 16 Menú de jPOS-PM-Test. Ejemplo 2

4.2.2. Entidades

Como se ha mencionado anteriormente, las entidades son la representación visual de las clases del modelo de negocio. Cada entidad se configura en un archivo xml particular y luego se agrega al archivo de configuración general del QBean del jPOS-PM.

Se recomienda que los archivos de entidades sean colocados en la carpeta cfg/entities en el módulo ui del proyecto.

La definición general de una entidad se describe a continuación.

```
<?xml version='1.0' ?>
<!DOCTYPE entity SYSTEM "cfg/entity.dtd">
<entity  id="anentity"  clazz="org.jpos.ee.SomeClass"  extendz="otherentity"
no-count="false">
    <dataAccess class="org.jpos.ee.pm.editor.DataAccessEntity" />
    <listFilter class="xxx" />
    <order>...</order>
```

```
    <highlights>...</highlights>
    <operations>...</operations>
    <field.../>
</entity>
```

Propiedad	Descripción
id	Id de la entidad. Debe ser único.
clazz	Clase del modelo de negocios que estamos representando.
extendz	Id de otra entidad de la cual se heredan los campos. (opcional)
no-count	Si se define como “true”, ninguna operación hace uso del método count del DataAccess. Se utiliza en entidades con numerosas instancias por cuestiones de eficiencia.
dataAccess	Sobreescribe el DataAccess por defecto definido en el archivo del QBean.
operations	La lista de operaciones disponibles para esta entidad.
field*	La lista de campos de esta entidad.
listFilter	El nombre de la clase de una implementación de la interface EntityFilter para filtrar el contenido de la lista de instancias. Requiere una revisión ya que debiera ser una propiedad de la operación listar.
order	String con los identificadores de los campos de esta entidad (y cualquier campo heredado) que determina el orden en el que aparecen los distintos campos en cada operación. Cualquier campo que no se incluya en este string, se envía al final.

Tabla 6. Propiedades generales de archivo de entidad.

El nombre descriptivo de la entidad es determinado por una entrada del archivo de internacionalización:

```
pm.entity.entityid=Nombre de la entidad
```

4.2.2.1. Operaciones

Las operaciones son las acciones que pueden realizarse sobre una instancia de una entidad. Por ejemplo, se pueden listar todas las instancias, agregar o editar una, etc.

Cada entidad posee sus propias operaciones y cada operación puede tener una configuración general o específica. A continuación se presenta la forma general en la que se definen las operaciones en un archivo de entidad.

```
<entity id="..." clazz="...">
<operations>
        <operation id="xxx" scope="xx" display="xx" enabled="xx" confirm="xx">
            <context class="xxx" />
            <properties>
                <property name="xxx" value="xxx"/>
                ...
            </properties>
```

```
        </operation>
        ...
    </operations>
    ...
</entity>
```

Propiedad	Descripción
id	Identificador único dentro de la entidad. Una misma operación puede utilizarse en más de una entidad pero sólo una vez en una misma entidad.
scope	item | general . Ítem indica que se afecta a una instancia en particular. General indica que la operación afecta a 0 o N instancias.
display	String con los identificadores de otras operaciones en las cuales la operación que se está definiendo aparecerá.
enabled	true | false . Determina si la operación está habilitada o no. Las operaciones deshabilitadas no están disponibles. Tiene el mismo efecto que borrar la operación pero sin perder su definición (para usos futuros).
confirm	true | false . Determina si se requiere de una confirmación antes de la ejecución de la operación. Útil principalmente en operaciones de un sólo paso (por ejemplo, borrar).
properties	Una lista de propiedades particulares para cada operación.
context	Ver a continuación.

Tabla 7. Propiedades generales de una operación

Cada operación puede tener un contexto de ejecución asociado (OperationContext). El contexto de una operación es una interface que define 3 métodos.

```
public void preConversion (PMContext ctx) throws PMException;
public void preExecute    (PMContext ctx) throws PMException;
public void postExecute   (PMContext ctx) throws PMException;
```

El método **preConversion** se ejecuta antes de realizar cualquier cambio a la o las instancias que se estén afectando con la operación.

El método **preExecute** se ejecuta luego de la modificación de la o las instancias pero antes de su persistencia, es decir, antes de la ejecución del DataAccess.

El método **postExecute** se ejecuta luego de la persistencia.

> **Nota**: El módulo pm_struts implementa una operación como un action de struts. Cada action tiene un parámetro llamado “pmid” que refiere a la identificación de la entidad.

i) Operaciones predefinidas

- **list**. Lista las instancias de la entidad. Normalmente es la pantalla principal del resto de las operaciones. Tiene las siguientes propiedades:
  - **paginable**: Agrega la habilidad de dividirse en páginas.
  - **searchable:** Agrega búsquedas locales dentro de una página.
  - **show-row-number**: Agrega el número de fila a cada instancia.
  - **sort-field**: Id del campo para el ordenamiento por defecto.
  - **sort-direction**: asc / desc
- **show**. Muestra el contenido de una instancia
- **add** Agrega una nueva instancia
- **edit** Modifica una instancia existente
- **delete** Borra una instancia
- **sort** Aplicable sólo a la lista, la ordena por alguno de los campos
- **filter** Aplicable sólo a la lista, la filtra con ciertos criterios

> **Nota:** Aunque normalmente cuando se opera sobre una instancia, ésta se recupera desde la lista (y no requiere de identificación especial, simplemente se accede por índice), también es posible acceder a instancias a través de una identificación utilizando el parámetro *identified=campo:valor*

A continuación se muestran algunas de las operaciones predefinidas en el contexto de otras operaciones.

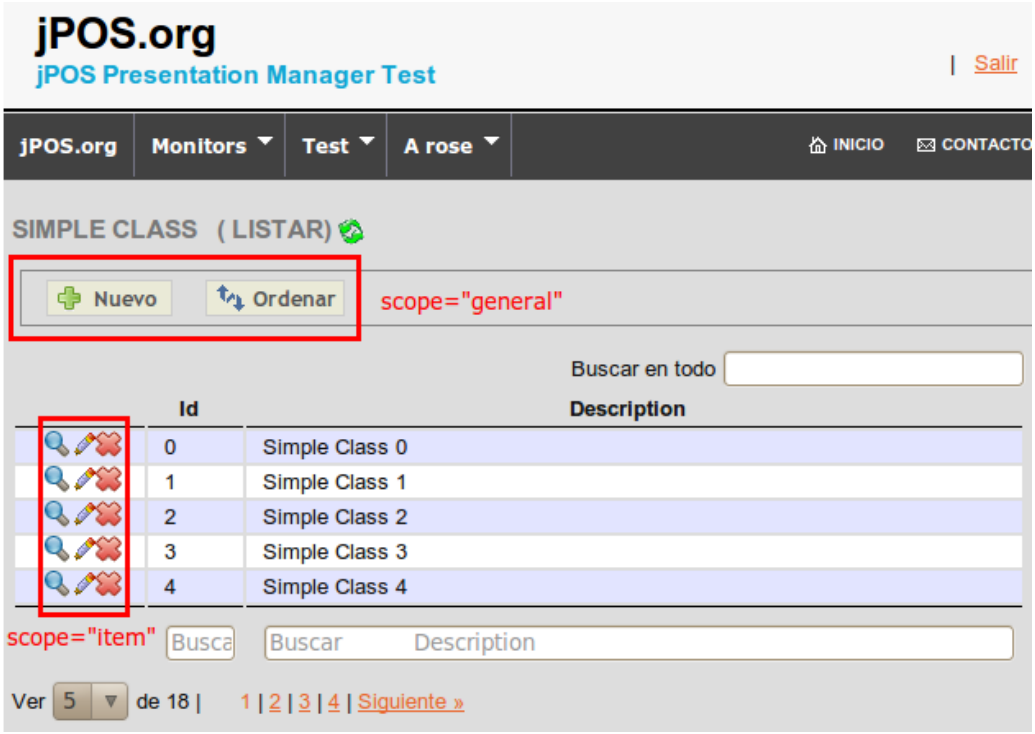

Figura 17 Operación "list"

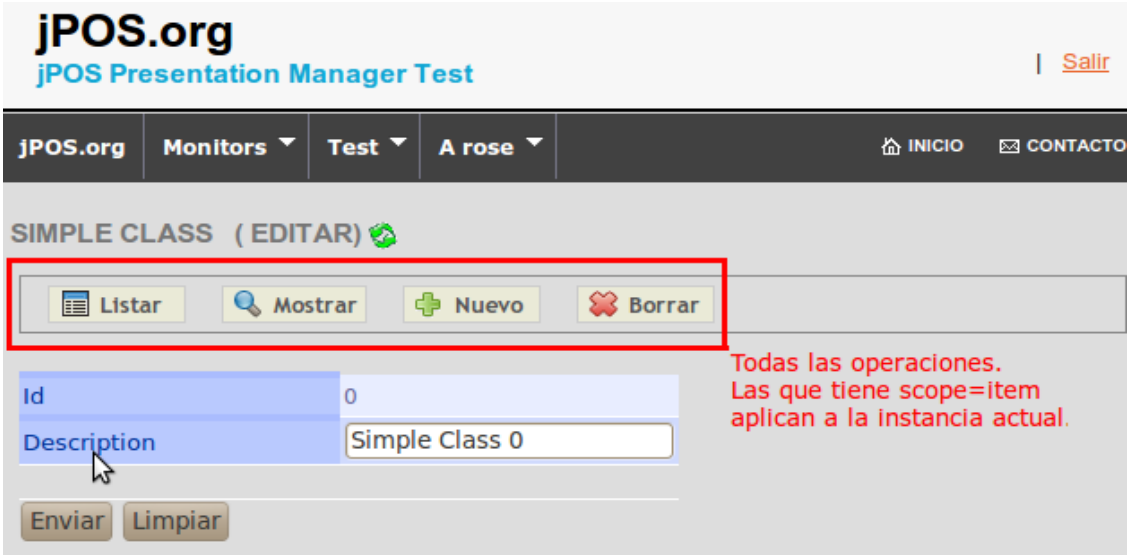

Figura 18 Operación "edit"

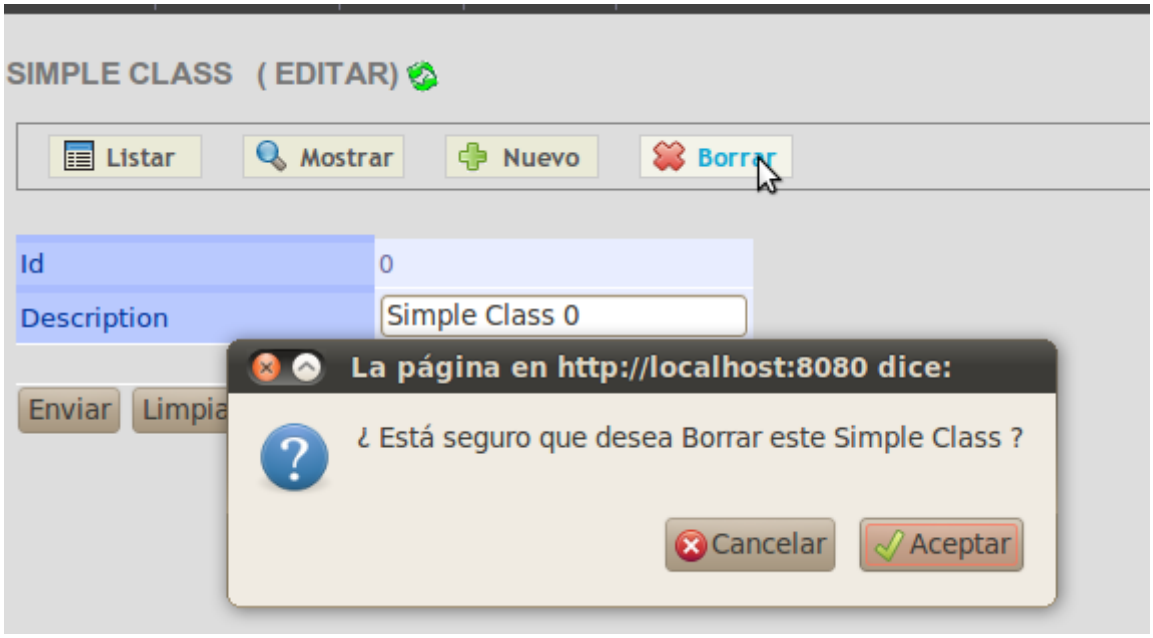

Figura 19 Confirmación de una operación

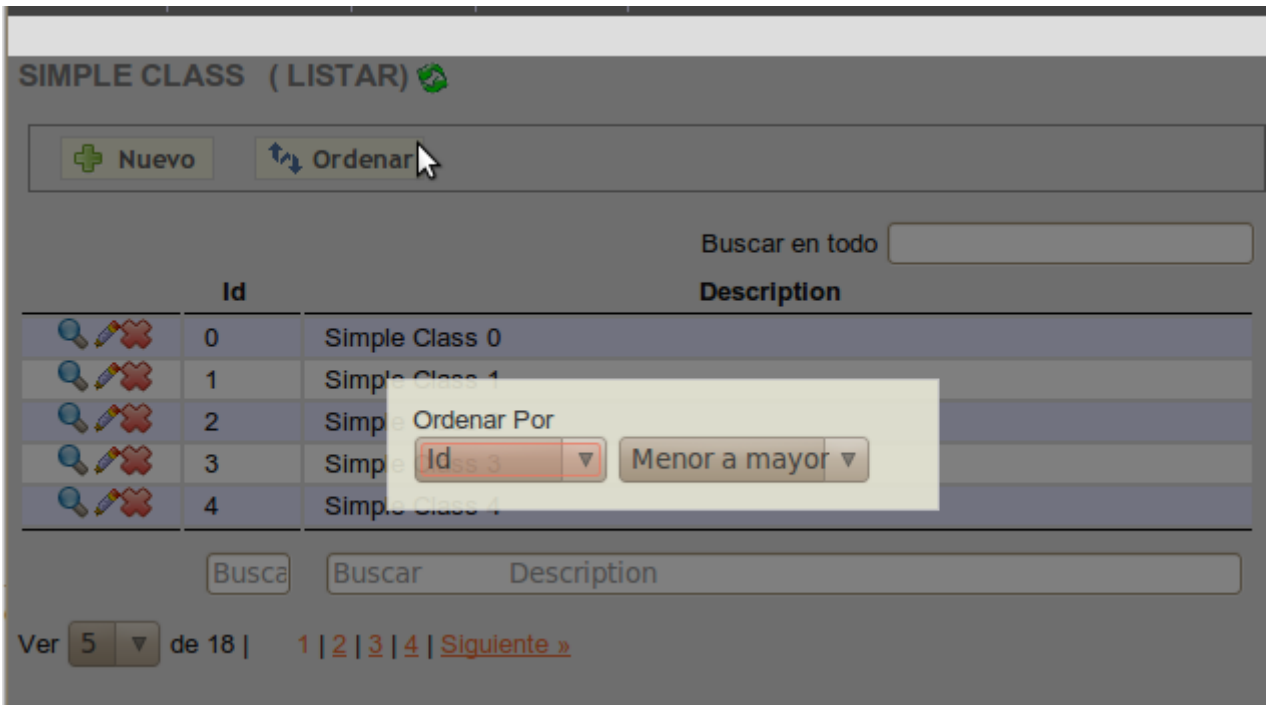

Figura 20 Operación "sort"

ii) Operaciones personalizadas

Además de las operaciones predefinidas, se pueden definir cualquier tipo de operación que actúe sobre una entidad (o no). Teniendo en cuenta que a nivel de pm_core una operación no es más que una definición y que a nivel de pm_struts las operaciones son actions, aplicando un simple procedimiento se pueden definir nuevas operaciones. En esta sección se describirá brevemente la metodología para agregar una nueva operación (sin entrar en detalle).

Suponiendo que se quiere definir una operación llamada "export". Los pasos a seguir son:

1. Definir el action.
2. Definir la visualización por medio de una página JSP (optativo).
3. Asignar un ícono.
4. Asignar un nombre en el archivo de internacionalización.

Para definir actions, beans y forwards en struts, el módulo pm_struts permite agregarlos a través de los archivos

```
cfg/struts.actions
cfg/struts.beans
cfg/struts.fwds
```

Por ejemplo, para agregar la operación export, se debe agregar el action al archivo struts.actions

```
cfg/struts.actions
------------------------------------------------------------------------
   <action path="/export" type="org.jpos.ee.pm.struts.actions.ExportAction">
      <forward name="success" path="/pages/export.jsp"/>
   </action>
```

El módulo pm_struts ofrece diversas clases para hacer heredar a los actions nuevos. Cada clase posee métodos de ayuda e integración con el resto de las operaciones.

Al momento de definir la página JSP de visualización (si se requiriera), se pueden hacer uso de varios tags JSP personalizados del módulo pm_struts, del framework Struts y de las librerías JSTL. También está disponible para uso las librerías javascript jQuery y jQuery-ui.

A cada operación debería asociarsele un icono descriptivo. Esto puede realizarse agregando un archivo gif transparente de 16x16 píxeles ubicado en la carpeta:

```
modules/project_ui/webapps/pm/templates/mytemplate/img/export.gif
```

Para que esté disponible dicha imagen en los botones, debe agregarse el siguiente estilo

```
modules/project_ui/webapps/pm/templates/mytemplate/buttons.css
--------------------------------------------------------------------------------
.export {
    background:url(img/export.gif) no-repeat 10px 8px;
}
```

Como último paso, debe definirse en los archivos ApplicationResource el nombre de la operación bajo la clave “operation.export”.

El resultado, dependiendo del scope de la operación se verá como muestra la **Figura 21** .

Figura 21 Operaciones personalizadas

4.2.2.2. Elementos resaltados (Highlight)

Un highlight es una declaración dentro de una entidad que permite resaltar ciertas instancias con colores. La versión actual de jPOS-PM tiene un resaltado limitado pero se espera extender la funcionalidad a futuro.

La definición general de la lista de highlights en un archivo de entidad es:

```
<entity id="..." clazz="...">
  <highlights>
    <highlight field="xxx" value="xxx" color="xxx" scope="xxx" />
  </highlights>
</entity>
```

Propiedad	Descripción
field	Id del campo por el cual se verificará la condición de resaltado.
value	Valor que debe tener el campo para que aplique el resaltado.
color	Color (en formato HTML).
scope	“instance”: aplica a toda la instancia. “property”: aplica solamente al campo.

Tabla 8. Propiedades de un highlight

Por ejemplo, en el jPOS-PM-Test la entidad complexclass2 tiene la siguiente definición de resaltados con el resultado que puede verse en la **Figura 22** .

```
    <highlights>
        <highlight field="id" value="3" color="yellow" scope="instance" />
        <highlight field="id" value="4" color="blue" scope="property" />
    </highlights>
```

Figura 22 Elementos resaltados

4.2.2.3. Entidades débiles

Una entidad débil es la representación de un objeto fuertemente dependiente de otro (composición). Normalmente el padre posee una lista de los objetos hijos.

Una entidad débil se define en el archivo de configuración de la siguiente forma.

```
<entity id="..." clazz="...">
    <owner>
        <entityId>xxxxx</entityId>
        <entityProperty>xxxxx</entityProperty>
        <entityCollectionClass>java.util.ArrayList</entityCollectionClass>
        <localProperty>xxxxx</localProperty>
    </owner>
    ....
</entity>
```

Propiedad	Descripción
entityId	El id de la entidad fuerte.
entityProperty	Propiedad de la entidad fuerte que contiene la colección de instancias de la entidad débil.
entityCollectionClass	Clase de la colección de instancias de la entidad fuerte.
localProperty	Opcional. Propiedad de la entidad débil que referencia a la entidad fuerte.

Tabla 9. Propiedades de la definición del padre en una entidad débil

La visualización de una entidad débil depende de una entidad fuerte, por esto, al realizar operaciones sobre una instancia, también puede verse afectada la entidad fuerte. La **Figura 23** y la **Figura 24** muestran como se destaca esta dependencia.

Figura 23 Entidad débil: operación "list"

Figura 24 Entidad débil: operación "edit"

El nivel de anidamiento de entidades débiles / fuertes es arbitrario, por lo que una entidad débil podría ser la entidad fuerte de otra.

4.2.2.4. Campos

Los campos o "fields" son la representación visual de los atributos de una clase del modelo de negocio que sea propenso a ser mostrado. Por ejemplo si se tiene la siguiente definición de una clase:

```
public class SimpleClass {
    private Long id;
    private String description;
    ...
}
```

los campos serían "id" y "description".

Cada atributo que quiera ser incluido en la representación visual debe tener su "getter" y "setter".

La definición general de un campo dentro de un archivo de configuración de una entidad es la siguiente:

```
<entity id="..." clazz="...">
    <field id="xx" property="xx" display="xx" align="xx" width="xx">
        <converters>
        ...
        </converters>
        <validators>
        ...
        </validators>
    </field>
    ...
</entity>
```

Propiedad	Descripción
id	Identificación única del campo en la entidad.
property	Opcional. Es el atributo de la clase que se está representando que se quiere visualizar. Es posible el anidamiento de propiedades de objetos a través de la notación puntual. Por ejemplo, si una clase Persona tiene un atributo objeto de tipo Direccion y quisieramos en la representación visual de Persona tener un campo "numero de dirección" se representaría como "direccion.numero". Si no está definido, se asume como atributo el id del campo.
display	Opcional. Es un string separado por blancos que define la lista de operaciones en las cuales aparecerá el campo. Se puede utilizar la palabra clave "all" para denotar que el campo aparece en todas las operaciones (valor por defecto).
align	Opcional. Define el alineamiento del campo. Puede tener como valores: left (por defecto), center o right
width	Opcional. Determina el ancho de las columnas en aquellas operaciones que pudieran necesitarlo (normalmente "list").

Tabla 10. Propiedades generales de los campos

El nombre del campo se determina por la clave "**pm.field.entityid.fieldid**" en los archivos de internacionalización.

Cada campo tiene asociado una lista de uno o más convertidores (o converters). Cada convertidor es el encargado de dos tareas

1. Traducir la representación interna del tipo de un campo en una representación visual apropiada para una operación.

2. Traducir la representación visual de una campo en una operación al tipo interno de dicho campo.

Cada operación tiene una representación diferente de cada campo. Un String se representa distinto ante una operación "list" o "show" (que simplemente lo muestra tal cual es) o una operación "add" o "edit" (que debe tener una representación editable).

Por ejemplo, dada la siguiente clase del jPOS-PM-Test:

```
public class ComplexClass1 extends SimpleClass{
    private BigDecimal amount;
    private Date date;
    private Date datetime;
    private Boolean active;
    private String password;
    private String key;
    private Long size;
    ...
}
```

La representación de cada campo en una operación "show" se muestra en la **Figura 25** .

Figura 25 Ejemplo de convertidores en operación "show"

Sin embargo, en una operación de edición, la representación sería como se muestra en la **Figura 26** .

Figura 26 Ejemplo de convertidores en operación "edit"

Algo que no sucede con la operación "show" pero si con la operación "edit" es que la segunda se aplica en dos pasos, primero la visualización tal cual se muestra y luego la aplicación de dicha operación. Aplicar la operación consiste en

1. Traducir el texto en un objeto del tipo apropiado (por ejemplo, traducir "0,149" en el decimal 0,149. Esta es la segunda tarea del converter.

2. Asignarle el valor obtenido de la conversión a la instancia de la entidad.

3. Asentar el cambio en el DataAccess.

El módulo pm_struts se encarga de cada uno de estos pasos.

La definición general de un convertidor es la siguiente:

```
<field id="xx" ...>
    <converters>
        <converter class="xxx" operations="xxx">
            <properties>
                <property name="xxx" value="xxx" />
                ...
            </properties>
        </converter>
    </converters>
</field>
```

Propiedad	Descripción
class	Class of the converter
operations	A white separated list of operation ids where this converter will be used for this field.
properties	A group of specific properties for this converter.

Tabla 11. Atributos de los convertidores.

Cada convertidor puede tener ciertas propiedades particulares, pero en general, todos tiene las siguientes propiedades.

Propiedad	Descripción
pad-count	En caso de estar las tres propiedades definidas, juntas representan el rellenado de la representación en string. “count” especifica el tamaño total del string resultante, “char” especifica el carater de relleno y “direction” especifica hacia donde se rellena (left o right).
pad-char	
pad-direction	
prefix	Prefijo anexado al resultado.
suffix	Sufijo anexado al resultado.

Tabla 12. Propiedades generales de los convertidores

Tanto el módulo pm_core como el pm_struts proveen diversos convertidores integrados, pero un desarrollador podría definir sus propios convertidores heredando la clase Converter. La sección 3.6.1 de la guía rápida describe todas las opciones para definir nuevos convertidores.

4.2.2.5. Validadores

Los validadores o “Validators” son clases que verifican estados no deseados antes de ejecutar una operación. La versión actual de jPOS-PM soporta dos tipos de validadores (que respetan la misma interface Validator):

1. **Validador de operación**: se define a nivel de operación y se ejecuta luego de la conversión y antes del acceso al DataAccess. En general la validación puede afectar a más de un campo o a la entidad como un todo.

2. **Validador de campo**: se define a nivel de campo y se ejecuta luego de la conversión de dicho campo. En general son validaciones que afectan a un campo y que aplican en cualquier operación.

La definición general de un validador es la siguiente:

```
<field id="xx" ...>
<!-- <operation id="xx"> -->
    <validator class="org.jpos.ee.pm.validator.XxValidator">
        <properties>
            <property name="xxx" value="xxx" />
        </properties>
    </validator>
</field>
<!-- </operation> -->
```

Cada campo u operación puede tener más de un validador y cada uno puede tener una o más propiedades. Normalmente, un validador tiene una o más propiedades que definen el mensaje a mostrar en caso no cumplirse con la validación.

Por ejemplo, el jPOS-PM-Test posee un validador de operación que verifica la unicidad del id:

```
<operation id="add" scope="general" display="all" >
 <validator class="org.jpos.ee.ui.validators.SimpleClassUniqueIdValidator">
     <properties>
         <property name="msg" value="pm.validator.simple.class.uniqueid" />
     </properties>
 </validator>
</operation>
```

Figura 27 Ejemplo de validador.

4.2.2.6. Herencia de entidades

Las entidades pueden definir una entidad de la cual heredar sus campos a través del atributo "extendz" del archivo de configuración. Esto tiene como efecto la inclusión de todos los campos de la entidad heredada excepto aquellos que ya están definidos (por equivalencia de id) en la entidad.

La herencia de campos incluye la definición de los convertores y validadores de campos.

La versión actual no permite herencia de operaciones ni de resaltados, funcionalidad que se espera incluir en un futuro.

4.2.3. Monitores

4.2.3.1. Definición

Los monitores son una forma de observar el estado de un origen de datos, tanto estáticos (como puede ser una tabla que no varía en cantidad de contenido sino en valores) como dinámicos (como puede ser un archivo de log el cual se incrementa periódicamente).

La definición de un monitor se realiza a través de un archivo de configuración xml que se recomienda esté ubicado en la carpeta “cfg/monitors” del módulo ui. Al igual que las entidades, los monitores deben incluirse en el archivo de configuración del QBean del jPOS-PM.

```
deploy/80_pm.xml
------------------------------------------------------------------------
    ...
    <property name="monitor" value="cfg/monitors/q2.xml" />
    ...
```

Un monitor es un elemento observable que periódicamente revisa su origen de datos para obtener información fresca. La información que obtiene el monitor se representa en forma de líneas que cada origen de datos retorna en un formato determinado (strings, arreglos, objetos, etc) y que son almacenadas dentro del monitor. El usuario puede luego, incluir observadores para el monitor, el cual informa de cambios. De esta forma, por más que haya muchos usuarios observando el mismo monitor, el origen de datos es solamente accedido desde un objeto. Esto evita la sobrecarga del origen de datos (que puede, por ejemplo, ser una tabla con muchos registros). Otra funcionalidad que evita la sobrecarga del origen es el echo de que el monitor consulta al origen sólo si tiene observadores.

La definición de un monitor tiene la siguiente estructura.

```
<?xml version='1.0' ?>
<!DOCTYPE entity SYSTEM "cfg/monitor.dtd">
<monitor>
    <id>monitorId</id>
    <source class="org.jpos.ee.pm.core.monitor.XxMonitorSource" >
        <properties>
            <property name="xx" value="yy" />
        </properties>
    </source>
    <formatter class="org.jpos.ee.pm.core.monitor.XxFormatter" >
        <properties>
            <property name="xx" value="yy" />
        </properties>
    </source>
    <delay>xx</delay>
    <max>xx</max>
    <cleanup>true | false</cleanup>
    <all>true | false</all>
</monitor>
```

Propiedad	Descripción
id	Identificación del monitor. Debe ser globalmente único.
source	Origen de los datos del monitor. Es una subclase de MonitorSource .
formatter	Subclase de MonitorFormatter (o incluso ella misma) que toma una linea del origen y lo transforma en un texto formateado para visualizar.
delay	Tiempo de espera entre refrescos al origen de datos.
max	Máximo numero de líneas que guarda el monitor.
cleanup	Si es “true”, cada obtención de datos del origen de datos borra lo anterior. “false” por defecto.
all	Si es “true”, cada acceso al origen de datos recupera toda la información (no solo la nueva). “false” por defecto.

Tabla 13. Propiedades de un monitor

4.2.3.2. Sources existentes

Hay dos clases predefinidas de orígenes de datos, las cuales se describen a continuación.

Módulo	Clase	Descripción
pm_core	org.jpos.ee.pm.core.monitor.FileMonitorSource	Origen de datos para observar un archivo. La propiedad “filename” es requerida e indica el archivo a observar.
pm_core_db	org.jpos.ee.pm.core.monitor.SQLMonitorSource	Origen de datos para observar el contenido de una consulta SQL. Tres propiedades son requeridas: • **query**: Consulta SQL para obtener toda la información. • **last-line-query**: Consulta SQL para obtener una nueva línea de información. • **id-column**: Índice de la columna que se utiliza como identificación (para determinar si existe información nueva).

Tabla 14. Orígenes o sources de datos predefinidos de un monitor

4.2.3.3. Formatters existentes

Hay dos clases predefinidas de formateadores, las cuales se describen a continuación.

Módulo	Clase	Descripción
pm_core	org.jpos.ee.pm.core.monitor.MonitorFormatter	Clase base de los formateadores. Recibe un objeto y retorna el método "toString()" de dicho objeto.
pm_core	org.jpos.ee.pm.core.monitor.BasicObjectArrayFormatter	Formateador que recibe un arreglo de objetos y los transforma en un string a través de las siguientes propiedades: • **pads**: Es un string separado por "#" que define el rellenado de cada ítem del arreglo (si es que lo tiene). Cada relleno tiene la forma "[W,X,Y,Z]" donde ◦ W=Índice del arreglo. ◦ X= R o L indicando la dirección de relleno (right o left). ◦ Y=Carácter de relleno. ◦ Z=Tamaño total del string resultante. • **separator**: Es un string que separa cada ítem del arreglo.

Tabla 15. Formateadores de monitores preexistentes.

4.2.3.4. Ejemplos

i) Monitor para el log por defecto de jPOS

```
<?xml version='1.0' ?>
<!DOCTYPE entity SYSTEM "cfg/monitor.dtd">
<monitor>
    <id>q2</id>
    <source class="org.jpos.ee.pm.core.monitor.FileMonitorSource" >
        <properties>
            <property name="filename" value="log/q2.log" />
        </properties>
    </source>
    <formatter class="org.jpos.ee.pm.core.monitor.MonitorFormatter" />
    <delay>3000</delay>
    <max>100</max>
</monitor>
```

El resultado se muestra en la **Figura 28** . La pantalla se actualiza automáticamente cuando hay nueva información en el log.

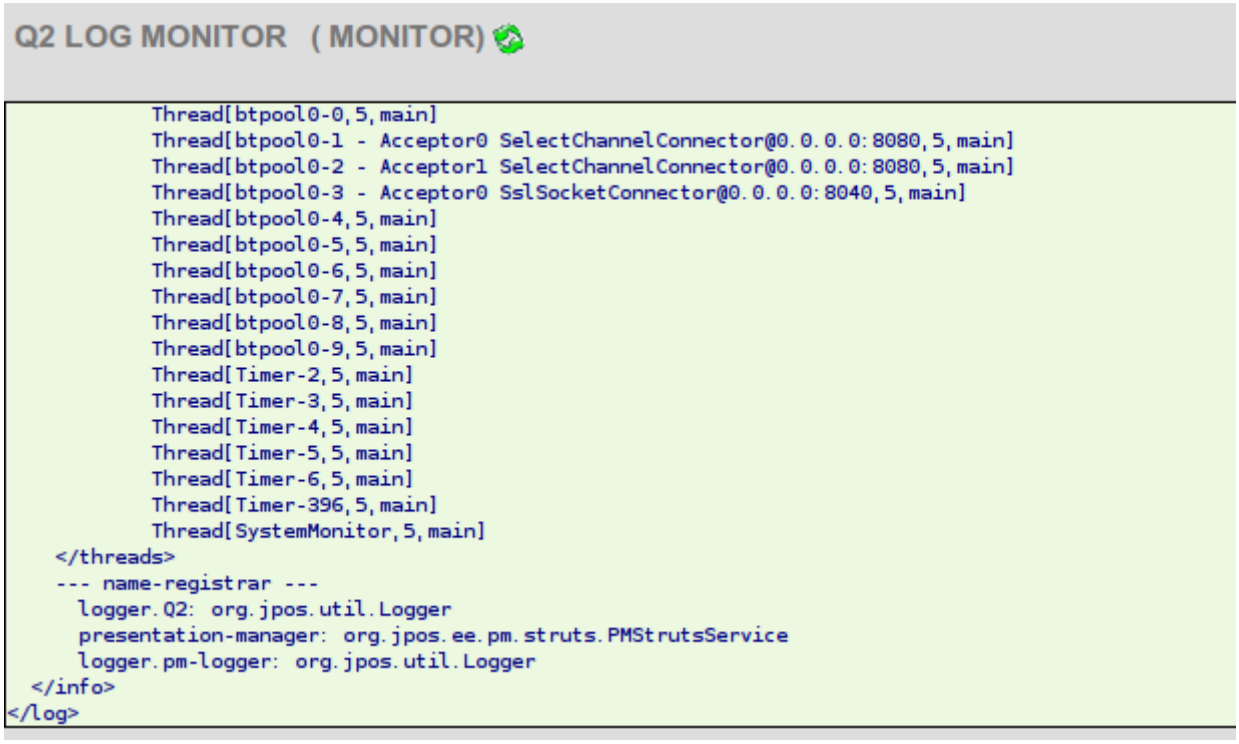

Figura 28 Pantalla del monitor del q2.log

ii) Monitor para la tabla “status” del módulo “status” de jPOS.

```
<?xml version='1.0' ?>
<!DOCTYPE entity SYSTEM "cfg/monitor.dtd">
<monitor>
    <id>status</id>
    <source class="org.jpos.ee.pm.core.monitor.SQLMonitorSource" >
        <properties>
            <property name="query"
                      value="SELECT id, name, state, detail, lastTick,
timeout, timeoutState FROM status" />
            <property name="last-line-query"
                      value="SELECT id, name, state, detail, lastTick,
timeout, timeoutState FROM status" />
            <property name="id-column" value="0" />
        </properties>
    </source>
    <formatter class="org.jpos.ee.pm.core.monitor.BasicObjectArrayFormatter">
        <properties>
            <property name="pads"
                      value="[0,R, ,20]#[1,R, ,40]#[2,R, ,2]#[3,R, ,75]"/>
            <property name="separator" value=" | " />
        </properties>
    </formatter>
    <delay>3000</delay>
    <max>100</max>
    <cleanup>true</cleanup>
    <all>true</all>
</monitor>
```

El resultado es una pantalla aparentemente estática pero que actualiza su contenido cuando hay cambios. Puede verse un ejemplo en la **Figura 29** .

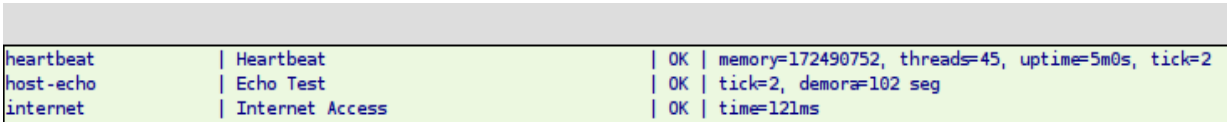

Figura 29 Pantalla del monitor de la tabla status

4.3. Aporte y ventajas

El jPOS-PM provee un amplio set de herramientas para el programador jPOS que le permiten desarrollar una interfaz administrativa en poco tiempo (utilizando las clases predefinidas) pero que a su vez le permite expandirla a su gusto gracias a la posibilidad de definir nuevas operaciones, convertidores, validadores, monitores y cualquier otra funcionalidad que se requiera, todo integrado a jPOS. Con esto se logra que el desarrollador se concentre en su sistema transaccional jPOS pero teniendo una interfaz de soporte rápidamente que le permite mostrar su trabajo al cliente, tarea difícil en este tipo de sistemas donde todo sucede “en las sombras”.

Otro aporte importante es la integración de las interfaces. Como se mencionó al comienzo de este informe, jPOS provee módulos dispersos con algunos tipos de visualización, pero nada completo e integrado. JPOS-PM provee un marco de trabajo para la definición de todo tipo de interfaces que, al estar basado en tecnologías conocidas (Web, JSP, Struts), permiten desarrollar cualquier tipo de interfaz que se requiera.

Por último, es importante destacar la independencia entre módulo principal y el módulo de visualización. Esta independencia permite definir todas las entidades y operaciones independientemente de la representación visual, lo que permite a los eventuales colaboradores del proyecto, desarrollar otro tipo de visualización distinta a la usada por defecto y cambiar de una a otra con algunos simples ajustes.

Capitulo 5. Resultados y conclusiones

5.1. Resultados esperados y obtenidos

5.1.1. Aporte a la comunidad OpenSource

Este proyecto surgió como resultado de una necesidad observada a partir de la experiencia personal de usuarios desarrolladores de jPOS y como objetivo principal tiene facilitar el trabajo del desarrollador y brindar al mismo tiempo potencia y flexibilidad en el desarrollo de interfaces.

Todo el código y la documentación producida, tanto del producto como del test están disponibles en un sitio de alojamiento libre y público de alta difusión (GitHub) y se espera que pronto esté disponible en el alojamiento oficial de módulos optativos de jPOS.

Estos dos puntos hacen que todo el proyecto se centre en realizar un aporte útil a la comunidad OpenSource de jPOS y motivar a su vez la formación de una nueva comunidad en torno a este proyecto.

Como ayuda a los desarrolladores interesados, se realizaron dos tareas sumamente importantes: el módulo jPOS-PM-Test como ejemplo integral de las funcionalidades y la "*jPOS Presentation Manager Quick Guide"* como documentación troncal para el desarrollador.

5.1.2. Inclusión en un proyecto real

A lo largo del proceso de desarrollo, el producto se incluyó, en una etapa temprana, en un proyecto de software privado contratado por una importante empresa chilena. Gracias a este proyecto, se descubrieron numerosas necesidades y problemas que se fueron resolviendo a lo largo de la evolución de ambos productos. Este proceso ayudó con la estabilidad y robustez del jPOS-PM y cumplimentó uno de los resultados esperados de este proyecto, utilizar el producto en al menos un proyecto real basado en jPOS.

Hoy en día el producto desarrollado utilizando jPOS-PM está satisfactoriamente instalado en producción y funcionando de forma estable.

5.1.3. Desarrollo final

Desde el punto de vista del software, se esperaba tener finalizada una versión inicial lo más completa posible que incluyera al menos, los requisitos planteados en la sección 3.3.1 de este informe. Dicho objetivo se cumplió con el desarrollo de los siguientes módulos finales:

- **pm_core**: módulo central que incluye las funciones principales y la definición de las clases fundamentales.

- **pm_struts**; módulo de implementación visual que ofreciera una "cara" a las funciones implementadas por el core.

- **pm_security_core**: módulo de seguridad básico integrado al resto y extensible a las necesidades del desarrollador.

- **pm_core_db** y **pm_security_db**: implementación de acceso a datos desde una base de datos relacional basada en hibernate para su uso en sistemas reales.

5.2. Trabajos futuros

Como framework de desarrollo, hay muchas tareas pendientes para realizar. Algunas de las planeadas para un futuro cercano se describen a continuación, pero como proyecto OpenSource se espera que surjan nuevas necesidades y funcionalidades periódicamente a partir de su utilización.

1. Nuevos módulos de implementación de interfaces basados en, al menos, Struts 2, Swing y JSF.

2. Módulo de auditoría.

3. Nuevos módulos de seguridad para autenticación basada en LDAP, Web Service y archivos simples.

4. Más independencia y más facilidades para la definición y uso de los convertidores.

5. Desarrollo de un editor para los archivos de configuración. Este proyecto ya está en desarrollo y está hecho en el propio jPOS-PM, aunque en una versión muy temprana y reducida.

6. Nuevos y mejores convertidores y validadores.

7. Operaciones para exportar listas.

8. Herencia de operaciones y resaltados.

9. DataAccess genérico para archivos xml arbitrarios.

10. Nuevas plantillas con mejor diseño gráfico.

11. Rediseño de la visualización de monitores.

5.3. Conclusión final

Este proyecto ha resultado altamente enriquecedor a nivel personal y profesional tanto desde el punto de vista de las tecnologías utilizadas como desde el punto de vista de la integración al mundo OpenSource.

Ha resultado gratificante la utilización satisfactoria del resultado en al menos un proyecto real y el uso y aceptación por parte de varios desarrolladores.

Por último, resulta motivador la posibilidad tangible de incluir el resultado al proyecto jPOS y la amplia gama de funcionalidades que pueden agregarse en versiones futuras.

Capitulo 6. Glosario

- **Campo**: representación visual de una variable de instancia de una clase del modelo de negocios.

- **Convertidor**: objeto asociado a un campo y una operación que determina la forma en la que un campo es transformado para ser visualizado y como de la representación visual es transformado al tipo original.

- **DataAccess:** interface de acceso al origen de datos del modelo de negocios.

- **Entidad débil**: representación de una clase del modelo de negocio que depende fuertemente de otra, es decir que no existe independientemente.

- **Entidad**: representación visual de una clase del modelo de negocio.

- **Formateador de monitor**: clase que dada una línea de información provista por un monitor, produce un String formateado de acuerdo a ciertos parámetros.

- **Instancia de entidad**: objeto instancia de una clase del modelo de negocios representada a través de una entidad.

- **Monitor**: elemento observable que obtiene regularmente líneas de información de un origen de datos e informa a sus observadores cuando hay cambios.

- **Operación:** representación de una acción realizable sobre cero, una o más instancias de una entidad.

- **Origen de monitor**: origen de las líneas de información de un monitor.

- **Resaltado**: representa una condición asociada a un campo por la cual una instancia de una entidad podría sobresalir por sobre otras en ciertas operaciones.

- **Validador**: objeto asociado a una operación o a un campo dentro de una entidad que se encarga de realizar validaciones previas a la persistencia al DataAccess.

Capitulo 7. Anexos

7.1. Anexo I: Cuestionarios

7.1.1. Mark Salter

Introducción

Mark Salter es un colaborador experto del proyecto jPOS. Es además, el administrador de la lista jpos-users@googlegroups.com, dedicada al debate de ideas y nuevas funcionalidades de jPOS.

Cuestionario

P: On how many projects have you worked using mainly jPOS ?

R: Many small projects have benefited from my application of jPOS, particularly in my QA and testing associations. I have recently worked on a vary large project using jPOS. This project uses jPOS, so would not count as one using mainly jPOS.

P: Can you briefly mention the "business" touched by some of this projects?

R: My use is mainly credit card processing Issuer and Acquirer. As mentioned above primarily as a tool to support, provide and drive testing of mainframe and 'java' based systems.

P: How many of these projects had a complex business model asociated?

R: Just the single large one - of which jPOS forms a small – but significant - part.

P: How many of these projects needed an extra development to build an administrative UI to handle the busisness model?

R: This large one did. My intial hope was that we would use jPOS (including UI components) exclusivly, but the UI for the whole solution was written using flex and covers the whole Issuing solution.

P: Was this extra development a heavy or significative job during the project?

R: It was a big part, but through choice rather than neccesity (we chose not to use the jPOS components).

P: Do you think that a jPOS module to dynamic build interface (using xml and with jPOS philosophy) could be usefull to make this extra development easier or faster?

R: I think so.

7.1.2. Andy Orrock

Introducción

Andy Orrock es un referente en jPOS desde hace años. Es un usuario activo y colaborador del proyecto y publica regularmente artículos relacionados a las transacciones electrónicas y a jPOS en su blog[5].

Cuestionario

P: On how many projects have you worked using mainly jPOS ?

R: Approximately eight.

P: Can you briefly mention the "business" touched by some of this projects?

R: Let me describe the business of some of the projects

- Check cashing for Mexicans in the US (some of the money transferred to Mexico and then remaining money on the cards - we managed the cards)
- Managing the payments made by customers at a pharmacy chain of stores
- Managing the drug prescription 'adjudication' at a pharmacy chain
- Helping a car warranty business manage their cards and act as a transaction authorization

P: How many of these projects had a complex business model asociated?

R: Seven of the eight projects have complex models like that.

P: How many of these projects needed extra development to build an administrative UI to handle the busisness model?

R: We've traditionally tried to stay away from UIs because we're OLTP developers. Plus, it always brings into play a customer's "look and feel" and "standards committee' (yuck). But there was 1 project where we had to do it.

5 www.andyorrock.com

P: Was this extra development a heavy or significative job during the project?

R: Oh, yes. The UI was a small part of the overall project functionality-wise, but we ended up spending A LOT of time coding and testing it (and dealing with browser release level quirks and FF/IE differentials)

P: Do you think that a jPOS module to dynamic build interface (using xml and having jPOS philosophy) could be usefull to make this extra development easier or faster?

R: Definitely. Anything to facilitate the rapid building of UIs inside the jPOS framework is something we will strongly consider using. We would spend more time putting good UIs into each of our projects.

7.1.3. Alwyn Schoeman

Introducción

Alwyn es un desarrollador de la empresa S1 Corporation[6] con años de experiencia en jPOS y varios aportes realizados al proyecto.

Cuestionario

P: On how many projects have you worked using jPOS ?

R: 4 projects.

P: Can you briefly mention the "business" touched by some of this projects?

R:

Project 1: Use of Q2 container to update mobile applications on SIM cards.

Project 2: Use as mobile banking interface to a bank.

Project 3: Payment kiosk and server.

Project 4: Clearance gateway for Mastercard and Visa for a client.

P: How many of these projects had a complex business model asociated?

R: At least 2

P: How many of these projects needed an extra development to build an administrative UI to handle the busisness model?

R: Only 1 was developed, though it would have been more if the budget and time allowed for it.

6 http://www.s1.com/

P: Was this extra development a heavy or significative job during the project?

R: No.

P: Did you used jPOS modules/components or was an external development?

R: For the UI, it was external, PHP.

P: Do you think that a jPOS module to dynamic build interfaces (using xml and with jPOS philosophy) could be useful to make this extra development easier or faster?

R: Yes.

7.1.4. David Bergert

Introducción

David Bergert es un experto en transacciones electrónicas de todo tipo y con amplia experiencia en jPOS. Es además, un colaborador activo del proyecto. Cargo: "Technology and Development Director On-Line Strategies, Inc.".

Cuestionario

P: On how many projects have you worked using mainly jPOS ?

R:

1) Acquiring Switch Project - OLS.Switch - Financial and Non-Financial transaction sets - approx. 22 endpoints/applications (see attached doc OLS Interfaces)
2) Issuing Switch / Authorization Host - OLS.Switch - MasterCard MIP implementation and Bank Switch integrations
3) Non-Financial (Pharmacy Adjudication - NCPDP Switch)
4) Issuing Switch / Authorization Host - Debit Card implementation
5) Store Valued Host - both issuing and acquiring - for many College/University Stored Value programs
6) Acquiring Debit Switch for Payment Processor
7) Bridge type interface between Visa DEX/VAPs and a payment processor - pass-though with billing modules to count txn's
8) Bridge type handling 4 outbound interaces with a common XML interface to client
9) Various internal tools/utilities/simulators ;) too many to remember
10) Server to convert various payment terminal message formats to a format a payment switched required. ISO8583 to native switch converter

11) Server to support file based processing for host based capture and clearing and settlement
12) Authorization hosts that support "external rules based" authorization to customers rules engines for final authorization
13) SSL based payment gateway and interface for IP enabled payment terminals
14) Server to interface with various cash registers in the restaurant market place
15) others I don't recall ;)

P: Can you briefly mention the "business" touched by some of this projects?

R: See above for descriptions -- let me know if you have any questions - I'm happy to expand

P: How many of these projects had a complex business model asociated?

R: 75% of these did

P: How many of these projects needed extra development to build an administrative UI to handle the busisness model?

R: 50% of these did.

P: Was this extra development a heavy or significative job during the project?

I wouldn't call it heavy - it was burdensome and time consuming for various reasons - some being users having specific requirements, others because of the bulk of screens and different types of users, their views and permissions, others learning the jPOS web way -- JPublish , Velocity, bsh, etc. instead of other common more prevalent java web frameworks - at least in the case of the eeweb3 modules...

P: Do you think that a jPOS module to dynamic build interface (using xml and having jPOS philosophy) could be usefull to make this extra development easier or faster?

In some cases yes, in others no. a CRUD interface for many items would be very helpful - in other cases that require custom screens, or compliance with "design standards", or certain forms, screens -- I don't think a CRUD interface would be sufficient. Makes more sense in an admin type tool then one an end-user or cardholder would use.

Also in many cases our customer ask for interfaces or webservices or (scary as it is --) access to databases (we recommend replicated datastores) for their own UI and reporting work some have web development staff and choose to perform their own UI work.

Capitulo 8. Bibliografía

1. ISO 8583-1:2003 - Financial transaction card originated messages -- Interchange message specifications -- Part 1: Messages, data elements and code values. http://www.iso.org/iso/catalogue_detail.htm?csnumber=31628.

2. ISO 8583 - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/ISO_8583.

3. Developer Resources for Java Technology. http://java.sun.com/.

4. Lenguaje de programación Java - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_Java.

5. Apache Ant - Welcome. http://ant.apache.org/.

6. Hibernate - JBoss Community. http://www.hibernate.org/.

7. Hibernate - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Hibernate.

8. jetty - Jetty WebServer. http://jetty.codehaus.org/jetty/.

9. Lenguaje Unificado de Modelado - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Lenguaje_Unificado_de_Modelado.

10. Observer (patrón de diseño) - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o).

11. Composite (patrón de diseño) - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Composite_(patr%C3%B3n_de_dise%C3%B1o).

12. Definición de tipo de documento - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Definici%C3%B3n_de_tipo_de_documento.

13. ISO 639-1:2002 - Codes for the representation of names of languages -- Part 1: Alpha-2 code. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22109.

14. ISO 639-1 - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/ISO_639-1.

15. ISO - Maintenance Agency for ISO 3166 country codes - Lists of country names and code

elements. http://www.iso.org/iso/country_codes/iso_3166_code_lists.htm.

16. ISO 3166-1 - Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/ISO_3166-1.
17. Alejandro Revilla. jPOS.org. *jPOS.org*, 2010. http://www.jpos.org/.
18. Christian Bauer and Gavin King. *Hibernate In Action*. Hanning, 2004.
19. David Garlan and Mary Shaw. An introduction to Software Architecture. 1994.
20. Roger S. Pressman. Modelo de Proceso de Software. In *SOFTWARE ENGINEERING. A Practitioner's Approach. European Adaptation*. 19.
21. Roger S. Pressman. Modelos evolutivos del Proceso de Software. In *SOFTWARE ENGINEERING. A Practitioner's Approach. European Adaptation*. 23.